VISIT…

LANZAROTE Caliente.com

信息系统工程丛书

语义信息模型及应用

张维明 主编

肖卫东 黄凯歌 徐振宁 等编著





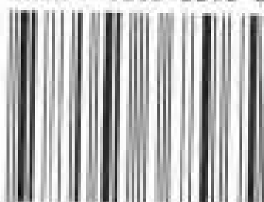
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

http://www.phei.com.cn

信息系统工程丛书

- 信息系统原理与工程
- 信息系统集成技术
- 语义信息模型及应用
- 信息系统建模
- 智能协作信息技术
- 多媒体信息系统
- 数据仓库原理与应用

语义信息模型及应用





策划编辑：秦 梅
责任编辑：秦 梅 李 萌
封面设计：闫欢玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-5053-6896-6/TP · 3922 定价：27.00 元

信息系统工程丛书

语义信息模型及应用

张维明　主编
肖卫东　黄凯歌　徐振宁　等编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

随着Internet的发展和信息共享需求的不断提升，语义信息模型必将成为Internet上的主流信息模型。作为语义信息模型的具体实现手段之一，XML将为许多问题提供崭新而有效的解决思路和方法。

本书试图从语义信息模型的基础知识入手，以XML为具体手段介绍语义信息模型及其应用。全书可分为两部分，第一部分由前4章组成，重点介绍语义信息模型和XML的基本知识，第二部分由后4章组成，介绍XML在人工智能、分布式计算、数据库和Web服务方面的应用。

本书既可作为高等院校信息管理相关专业硕士研究生课程的参考书，也可以作为广大科技工作者和研究人员的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，翻版必究。

图书在版编目(CIP)数据

语义信息模型及应用/张维明主编．—北京：电子工业出版社，2002.3
(信息系统工程丛书)
ISBN 7-5053-6896-6

Ⅰ．语… Ⅱ．张… Ⅲ．语义信息-模型 Ⅳ．G201

中国版本图书馆CIP数据核字(2002)第009457号

策划编辑：秦　梅
责任编辑：秦　梅　李　萌
印　　刷：北京东光印刷厂
出版发行：电子工业出版社　http://www.phei.com.cn
　　　　　北京市海淀区万寿路173信箱　邮编100036
经　　销：各地新华书店
开　　本：787×1092　1/16　印张：13.25　字数：339.2千字
版　　次：2002年3月第1版　2002年3月第1次印刷
印　　数：4 000册　定价：27.00元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。
联系电话：(010)68279077

《信息系统工程丛书》编委会

主任委员 郭桂蓉 总装备部科技委副主任，中国工程院院士，教授

副主任委员 卢锡城 国防科技大学副校长，中国工程院院士，教授

编委 高小山 中科院系统科学研究所副所长，研究员
怀进鹏 北京航空航天大学副校长，教授
钟玉琢 清华大学计算机系教授，中国计算机学会多媒体专委会主任
张维明 国防科技大学管理学院副院长，教授
文宏武 电子工业出版社副社长

执行秘书 秦 梅 肖卫东

《语义信息模型及应用》编写人员名单

主编 张维明

编著 肖卫东 黄凯歌 徐振宁 汤大权 曹泽文 李勇 李由 宋剑峰

丛　书　序

从现实世界的角度看，客观世界是由物质、能量和信息三大基本要素组成的，人类的社会生活每时每刻都离不开信息。从远古时代开始，人类就一直在同信息打交道，围绕着信息形成了不同的信息作业，包括信息的采集、存储、表示、传递、加工处理、检索利用和控制等，所有这些环节形成了信息系统，并作为客观世界每个系统的一个子系统或显式或隐式地存在着。

从科学技术的角度看，信息系统是在20世纪中叶由信息科学、计算机科学、管理科学、决策科学、系统科学等学科相互渗透交叉而发展起来的，经过多年的研究目前已经形成了比较完整的独具特色的体系。信息系统工程是20世纪80年代出现的以建立信息系统为目标的新兴学科，它是用系统工程的原理、方法来指导信息系统建设与管理的一门工程技术学科，主要研究各级各类信息系统建设和管理中的规律性的问题。它既不是“信息的系统工程”，也不是“信息系统的工程”，而是“信息系统的系统工程”。一般认为，信息系统工程的目标是为以计算机和其他信息技术为手段的各类信息系统提供科学的开发方法、管理手段及有关的工具、标准、规范，通常不包括通信工程、信号处理等具体学科领域的技术。

信息系统工程的研究范围主要包括：

(1)信息系统的基本理论。信息系统的基本观点、认识论和方法论等。

(2)信息系统建模。信息系统概念模型、逻辑模型和物理模型的描述、观察、试验与验证等。

(3)信息系统开发。信息系统建设与管理的概念、方法、评价、规划、工具、标准等一系列相关技术问题和工程问题。

(4)信息系统支撑技术在信息系统中的应用。数据库/数据仓库、网络通信、人机交互、分布计算、决策支持、人工智能等技术如何满足信息系统各层次用户的需求，实现业务管理、信息共享、分析决策等功能，并在组织和人的参与下最终达到信息系统的目标。

(5)信息系统集成。研究系统集成的原则、方法、技术、工具和有关的标准、规范，应用先进的相关技术，将支持各个信息“孤岛”的小运行环境，集成统一在一个大运行环境中，最终形成一体化的信息系统。

《信息系统工程丛书》是由国防科技大学管理学院组织多位专家和科研人员面向信息系统工程专业撰写的教材类图书。作者所在的单位是20世纪70年代末在钱学森院士的亲自倡导下建立起来的，在国内最早开设了信息系统工程专业。作者长期从事信息系统工程方面的教学、科研和开发，这套丛书是其多年学术研究和科技开发的成果总结，也是其多年教学工作中的实践积累，从丛书体系的设置到内容的安排，都基本体现了对当今信息系统工程领域前沿技术的把握。

这套丛书准备分批出版，第一批由《信息系统原理与工程》、《信息系统集成技术》、《信息系统建模》、《多媒体信息系统》、《智能协作信息技术》、《数据仓库原理与应用》、《语义信息模型及应用》7部教材和专著组成，再加上该单位近年已出版的《决策支持系统技术》和《智能决策支持技术》两部研究生教材，基本上已覆盖了上述的信息系统工程主要研究范围。其中：

《信息系统原理与工程》主要介绍信息系统的基本概念、基本原理、技术和设计开发方法。

具体包括信息系统与信息系统工程的基本概念，信息系统中的基础理论、开发方法，结构化系统分析、系统设计和面向对象的分析设计方法，信息系统战略规划，系统实施，信息系统对计划、控制、决策的支持，计算机辅助信息系统开发等。

《信息系统集成技术》主要介绍信息系统集成的基本概念、基本原理和设计开发方法。首先介绍信息系统集成技术的发展，然后从体系结构入手，分网络集成、数据集成和应用集成三个层次展开对信息系统集成的论述，并给出了系统集成的案例。

《信息系统建模》主要介绍信息系统建模的基本概念、基本原理、方法、工程技术与工具。具体包括面向信息系统建模的思想，需求建模，逻辑建模，对象建模，Agent 建模，数据建模，统一建模语言等，是国内第一部按照较完整的体系专门介绍信息系统建模技术的著作。

《多媒体信息系统》主要介绍多媒体信息系统的基本概念、原理、技术和应用，主要内容包括多媒体信息系统的体系结构和数据模型、多媒体数据库和信息管理、多媒体通信和网络、多媒体人机交互与表现技术、原型系统与应用等。

《智能协作信息技术》主要介绍智能协作信息技术及系统的基本概念、基本原理和设计开发方法。具体包括智能协作信息技术的发展概况，智能主体概念、性质、内部结构和实现方法，多智能主体协作的基本原理、实现技术等，还介绍了智能协作信息系统的开发方法和智能协作信息技术在工业、管理、办公自动化等领域的应用。它是国内第一部全面介绍智能协作信息技术和智能协作信息系统的专著。

《数据仓库原理与应用》主要介绍数据仓库的概念、基本原理、规划、开发方法以及相关算法，包括数据仓库的发展、技术体系、元数据管理、分析设计方法和开发工具，并对数据开采的主要理论和方法、联机分析等应用技术作了深入的阐述，是一本理论与实践相结合的教材，是国内较为全面地分析数据仓库、开发数据仓库的书籍。

《语义信息模型及应用》深入到目前信息管理领域的前沿，探讨了语义信息模型的基本概念，并以 XML 为具体实现手段介绍了语义信息模型在信息组织、信息处理、信息服务、信息交换等方面高级应用的原理与实现机制。

除《语义信息模型及应用》以外，丛书中所有教材都作为内印教材或讲义试用过多次，吸收了许多专家学者以及学生的意见。

这套丛书既能够使广大读者从整体上把握知识结构、理清相关技术领域的关系和分类，又能够从中找到每项具体理论、技术、方法、工具的介绍和例解，再加上融合了多项“九五”期间的高水平科研成果，应该使这套丛书具有较高的系统性和实用性。

《信息系统工程丛书》是一套理论与工程实践并重的著作，它不仅可作为相关专业的大学本科生和研究生的系列化教材和参考书，而且也可以为从事信息系统工程的科研人员提供参考。我们相信，这套丛书的出版，将对我国信息系统工程的全面、深入发展起到重要的推动和促进作用。

《信息系统工程丛书》编委会

2001 年 6 月

前　言

在整个人类文明发展的历史上，对话、交流与理解的历程一直伴随着信息技术的发展步伐。Internet 的产生和发展，使全人类的交流更加开放。从目前的状况来看，Internet 要想完全发挥其威力，不仅在于更快的 CPU 和更宽的带宽，还在于建立一种更有利于交流与对话的机制，在于发展一种更有利于相互理解的基础技术。这种技术必须从最基本的信息表示和交换开始，排除一切平台和语言的分歧，以自由、平等、开放为原则，以人类对现实世界的一致理解为基础，为全人类提供一种全新的高质量的信息服务。

在因特网发展过程中，有三大技术起了决定性的作用：第一是分组交换技术与中介信息处理器（Interface Message Processor）的发明，使分布式网络——APARnet（Internet 的前身）得以诞生；第二是 TCP/IP 协议的提出与实施，使 APARnet 扩展延伸，数据传输畅通无阻；第三是 HTML 与 WWW 的出现，使得一个全球最大的信息资源利用系统诞生了。我们仔细分析一下可以看出，分组交换技术与中介信息处理器的发明使得物理层的扩展成为可能，TCP/IP 协议的提出与实施在网络层统一了机器交互的语法。这两层属于信息基础设施，技术已趋成熟，不管怎样总是朝着超高带宽发展，但有一点是可以肯定的，那就是对应用层的透明性。目前，让 Internet 完全发挥威力的努力更多地针对应用层，而应用层要从根本上得到发展，只有走统一语义的道路。有了基于语义的信息表示、信息交换、信息处理技术，有了对标准化道路的一致共识，我们对于这个宏伟的目标就有了前所未有的信心。

全书共分 8 章，主要论述以 XML 为代表的语义信息模型及其在各个相关领域的应用。全书共分两大部分。第一部分由前 4 章组成。第 1 章概述语义、语义信息模型和 XML 的基本概念。第 2 章介绍语义信息模型的关键基础——资源描述框架 RDF。第 3 章探讨利用本体技术表示和处理语义的方法。第 4 章介绍语义 Web 的结构和思想，展现了语义信息模型在下一代 Web 中的应用。第二部分由后 4 章组成。第 5 章介绍 XML 在人工智能领域中的一些研究进展和成果。第 6 章介绍 XML 的语义消息在分布式计算中的应用。第 7 章分析 XML 与关系数据库的互相映射方法。第 8 章介绍 XML 在 Web 服务中的作用。

本书是作者多年潜心研究和开发工作的总结，是研究小组所有师生集体智慧的结晶。由于语义信息模型还正处在研究阶段，加之作者水平有限，难免在完整性、准确性等方面存在着问题。我们迫切希望能够得到广大读者特别是专家和同行的指点，共同探讨有关问题，交流研究经验，使我们的研究工作取得进步。

作　者

目　　录

第1章　语义信息模型概述 …………………………………………………………（1）
1.1　信息模型……………………………………………………………………（1）
1.2　语义信息模型………………………………………………………………（2）
1.2.1　什么是语义信息模型……………………………………………………（2）
1.2.2　语义信息模型的用途……………………………………………………（2）
1.2.3　语义信息模型的内容……………………………………………………（3）
1.2.4　语义信息模型的开发……………………………………………………（3）
1.3　XML与语义信息模型 ………………………………………………………（4）
1.3.1　Web与语义信息模型 …………………………………………………（4）
1.3.2　XML与语义信息模型 …………………………………………………（5）
1.4　XML与数据的语义表示 ……………………………………………………（5）
1.4.1　数据表示的第一次统一…………………………………………………（6）
1.4.2　XML的语义数据表示 …………………………………………………（7）
1.5　XML的应用进展 ……………………………………………………………（9）
第2章　资源描述框架(RDF) …………………………………………………（12）
2.1　概述…………………………………………………………………………（12）
2.2　RDF模型和语法 ……………………………………………………………（14）
2.2.1　基本RDF …………………………………………………………………（14）
2.2.2　容器…………………………………………………………………………（19）
2.2.3　关于声明的声明…………………………………………………………（23）
2.2.4　RDF正式模型 ……………………………………………………………（23）
2.2.5　RDF正式语法 ……………………………………………………………（24）
2.3　RDF模式(Schema) …………………………………………………………（25）
2.3.1　范围…………………………………………………………………………（26）
2.3.2　类和属性……………………………………………………………………（27）
2.3.3　约束…………………………………………………………………………（31）
2.3.4　可扩展性机制………………………………………………………………（33）
2.3.5　文档…………………………………………………………………………（34）
2.3.6　模型和语法概念……………………………………………………………（35）
2.3.7　例子…………………………………………………………………………（36）
2.4　Web数据集成的元数据解决方案 …………………………………………（38）
2.4.1　RDF实现Web元数据描述与交换的机制 ……………………………（38）
2.4.2　RDF的特点 ………………………………………………………………（39）
2.4.3　RDF与若干Web新技术 ………………………………………………（40）
2.5　借助RDF增强WSDL ………………………………………………………（41）

2.5.1 RDF 和 WSDL …… (42)
2.5.2 WSDL 的详细图形表示 …… (45)
2.5.3 用于 WSDL 的 RDF 模式 …… (45)
第 3 章 XML 的语义表示和处理方法——XML 与本体技术的结合 …… (48)
3.1 XML 与语义 …… (48)
3.2 XML 语义互操作问题 …… (49)
3.3 Ontology 理论 …… (50)
3.3.1 本体是什么 …… (50)
3.3.2 本体的基本内容 …… (51)
3.3.3 本体的研究现状 …… (51)
3.3.4 本体在互操作方面的应用 …… (52)
3.4 Ontology 的开发和应用 …… (54)
3.4.1 本体的开发方法 …… (54)
3.4.2 本体与 XML 结合的基本方法 …… (62)
第 4 章 语义 Web 的结构和思想 …… (80)
4.1 XML 下的 Web 的体系结构 …… (80)
4.2 Web 信息空间 …… (81)
4.2.1 HTTP 空间 …… (81)
4.2.2 内容和远程操作 …… (82)
4.2.3 数据格式 …… (83)
4.2.4 人类可读信息 …… (84)
4.2.5 机器可理解信息 …… (86)
4.2.6 元数据应用 …… (87)
4.3 语义 Web 的概貌 …… (88)
4.3.1 语义 Web 应用的设想 …… (88)
4.3.2 表达意思 …… (88)
4.3.3 知识表现 …… (89)
4.3.4 本体(Ontology) …… (90)
4.3.5 主体(Agent) …… (91)
4.3.6 知识的进化 …… (92)
4.4 Web 的设计原则 …… (92)
第 5 章 XML 在人工智能中的应用 …… (103)
5.1 人工智能的历史与现状 …… (103)
5.1.1 形成及第一个兴旺期(1956 年～1966 年) …… (103)
5.1.2 第二个兴旺期(20 世纪 70 年代中期～20 世纪 90 年代) …… (104)
5.1.3 第三个发展时期(20 世纪 90 年代末至今) …… (105)
5.2 人工智能的研究领域 …… (106)
5.3 XML 在人工智能中的应用 …… (107)
5.4 规则标记语言——RuleML(Rule Markup Languages) …… (108)
5.4.1 RuleML 基本概念 …… (108)

5.4.2 RuleML 实例 …… (112)
5.5 HornML(Horm Logic Markup Languages) …… (116)
第 6 章 XML 与分布式计算 …… (123)
6.1 分布式计算 …… (123)
6.1.1 分布式计算的优点 …… (123)
6.1.2 分布式计算的现有机制 …… (123)
6.1.3 CORBA 如何增强分布式计算 …… (124)
6.1.4 基于主体的分布计算模型 …… (124)
6.2 XML 与分布式计算 …… (125)
6.2.1 数据传递 …… (125)
6.2.2 接口管理 …… (127)
6.2.3 远程调用 …… (128)
6.2.4 统一的分布式系统体系结构 …… (131)
6.3 集成 XML 和 CORBA …… (131)
6.3.1 中间件技术 …… (132)
6.3.2 集成 XML 和 CORBA …… (132)
6.4 XML-RPC …… (136)
6.4.1 RPC …… (136)
6.4.2 RPC 和 XML …… (137)
6.4.3 XML-RPC 与 Java …… (138)
6.5 简单对象访问协议——SOAP …… (139)
6.5.1 简介 …… (139)
6.5.2 SOAP 消息交换模型 …… (141)
6.5.3 与 XML 的关系 …… (142)
6.5.4 SOAP 封装 …… (142)
6.5.5 SOAP 编码 …… (143)
6.5.6 在 HTTP 中使用 SOAP …… (144)
6.5.7 在 RPC 中使用 SOAP …… (146)
第 7 章 XML 与数据库 …… (148)
7.1 XML 与数据库的关系 …… (148)
7.2 XML 与数据库的结合类型 …… (148)
7.2.1 Native XML Database(NXD) …… (150)
7.2.2 XML-Enabled Database (XEDB) …… (150)
7.2.3 Hybrid XML Database(HXD) …… (150)
7.3 存储和检索数据 …… (151)
7.3.1 转移数据 …… (152)
7.3.2 从文档结构到数据库结构的映射类型 …… (152)
7.3.3 从数据库的结构生成 DTD 及其互逆过程 …… (154)
7.4 XML 数据库产品简介 …… (163)
7.5 SQL Server 2000 对 XML 存储的支持 …… (167)

7.5.1 SQL Server 的 HTTP 访问 …… (168)
7.5.2 使用带批注的 XDR 架构创建 XML 视图 …… (169)
7.5.3 检索并写入 XML 数据 …… (172)
7.6 Oracle8i 对 XML 存储的支持 …… (172)
7.6.1 Oracle Internet 文件系统(iFS) …… (173)
7.6.2 XML 基础结构组件之一 —— Oracle XML SQL Utility …… (174)
7.6.3 XML 基础结构组件之二 —— Oracle XSQL Servlet …… (176)
第 8 章 XML 与 Web 服务 …… (178)
8.1 什么是 Web 服务 …… (178)
8.1.1 Web 服务的计算模式 …… (179)
8.1.2 Web 服务对商业模式的转变 …… (179)
8.2 Web 服务的关键技术 …… (180)
8.3 Web 服务的体系结构 …… (181)
8.3.1 Sun ONE 软件体系结构 …… (182)
8.3.2 微软.NET 框架 …… (183)
8.4 智能化 Web 服务 …… (186)
8.4.1 对智能化 Web 服务的理解 …… (187)
8.4.2 智能化 Web 服务中的用户信息共享问题 …… (187)
8.4.3 为 Web 服务增加智能 …… (188)
8.4.4 智能化 Web 服务处理模式 …… (192)
8.5 Web 服务开发模式 …… (193)
8.5.1 基于 Java 的 Web 服务开发模式 …… (193)
8.5.2 微软.NET 开发模式 …… (195)
参考文献 …… (197)

第 1 章　语义信息模型概述

在整个人类文明发展的历史上，对话、交流与理解的历程，就是整个人类文明发展的历程；对话、交流与理解的程度，代表着人类文明发展的程度。

纵观 IT 业的发展历史，有一条由封闭走向开放的明确主线，开放的程度代表着 IT 业发展的水平。尤其是 Internet 的发展，使 IT 技术开放的步伐加大、加快了，改变了整个 IT 界的思路，“开放”的号角与“标准化”的鼓声响彻 IT 技术的每一个角落。

Internet 要完全发挥出其威力，不仅在于更快的 CPU，不仅在于更宽的网络带宽，还在于建立一种更有利于交流与对话的机制，在于发展一种更有利于相互理解的基础技术。这种技术必须从最基本的数据表示和信息交换开始，排除一切平台、语言的分歧，以自由、平等和开放为原则，以人类对现实世界的一致理解为基础，为全人类提供一种全新的高质量的信息服务。

有了基于语义的信息表示、信息交换和信息处理的技术，有了对标准化道路的一致共识，这些宏伟的目标正逐步变为现实。

1.1　信息模型

在过去数年里，“信息模型”这一术语频繁地出现在各种与信息技术相关的文档中。美国科学家联合会对“信息模型”的解释为：信息模型用于描述组织内的信息资源以及这些资源的相互关系。信息模型用于支持数据建模、生成数据库和文档存储的设计需求，提供数据体系结构的信息资源管理者视图（TAFIM 3.0）。

信息作为一种资源是人们已经逐步达成的共识。物质、能量和信息都是人类可以利用的战略资源，它们都可以用来制造生产工具。物质资源可以被加工为材料，但是仅利用材料就只能制造出人力工具。能量可以被转换为动力，它与材料结合起来就可能制造动力工具。而信息可以被提炼成为知识，它与动力和材料相结合，则可以制造出智能工具。人力工具可以支持农业社会的生产力，动力工具可以支持工业社会的生产力，而智力工具则可以支持信息社会的生产力。

在信息社会中，各种组织为了从容地应对频繁变化的环境，需要准确而迅速地掌握组织自身基础结构和环境究竟发生了哪些变化，这就必须要了解反映自身基础结构以及采取相应行动所必需的信息以及它们之间的关联。为此，组织内部成立专门的部门，开发大量的信息系统来解决这些问题。那么究竟要管理哪些信息资源呢？

“信息”以及“数据”的定义可谓仁者见仁，智者见智，加起来不下数十种。具体到信息系统中，参考 IDEF1X 的观点应该最具有实际指导意义。在美国，政府强制要求信息系统建设时必须采用 IDEF1X 方法进行数据建模。IDEF1X 方法认为“信息”是为了某个特定目的或在一定范围内聚合起来的数据集，而数据可以被认为是有含义事件的符号表示，单一的含义能够被用于多个不同的事件。

以上的定义实际上反映了对信息构成的一种理解，即信息是由数据构成的，而数据是由事实和其代表的含义构成的。同时也表明，建立信息模型管理信息资源的目的和重点应该放在

被管理数据应用在事件的含义上，或者说放在数据所代表的语义上。

1.2 语义信息模型

这一节我们对语义信息模型进行介绍。有些观点是我们在研究中逐步形成的。希望能和专家读者进行探讨。

1.2.1 什么是语义信息模型

我们认为数据库的三模式结构是外部模式、内部模式和概念模式。但是在过去，系统分析员们定义的模式结构大多只是外部模式和内部模式这两个模式，留下一个概念模式在自己的头脑中或不规范的记录中。

概念模式是数据的一个单一集成定义。在一个组织中，它不偏向于任何专门的数据应用，同时还独立于数据的物理存储和存取方式。这个概念模式的主要目标是提供一个数据的含义和相互关系的一致定义，从而用来集成、共享和管理数据的完整性。

从直观上讲，可能不会有人反对将数据库概念模式称为一种语义信息模型。但是同“信息模型”等基本术语一样，目前“语义信息模型”也没有一种公认的定义，甚至对它的定义也不多见。这种情况的出现并不意味着对“语义信息模型”的研究不重要，而可能是由于这样的一些考虑：任何信息模型都或多或少地描述或隐含了信息资源的语义，从而不需要再单独强调语义信息模型。但同时，我们都明确地知道概念模式、内模式和外模式是不同类型的模式，解决的问题不同，包含的语义不同等。实际上语义数据模型化技术的发展也开始于从概念视图定义数据的要求。

显然，语义信息模型有不同于一般信息模型的特征。归纳出这些特征，能够在一定程度上明确语义信息模型的内涵。语义信息模型必须具有以下的基本特征：

(1) 语义信息模型是与问题域的基础结构相一致，能够覆盖问题域当前应用范围的信息模型。

(2) 语义信息模型是明确地描述组织的基础结构，能够转化为多种视图和存储结构的信息模型。

(3) 语义信息模型是明确地记录对问题域基础结构进行概念化时所作的约定、假设和依据等的信息模型。

(4) 语义信息模型是可扩展的信息模型。

可以看出，这里所指的“语义”不同于语义学中的定义。从语义学的角度讲，语义是语言形式表达的内容。语义是思维的体现者，是客观事物在人们头脑中的反映，是人们交际过程的中心所在。从信息模型的角度讲，语义是构建在一定语法上，反映一定认知结果的数据对象，数据对象之间关系的描述与客观存在的一种对应关系。因此，信息模型中的语义与对客观存在的概念化以及描述认知结果的语言密切相关。定义这种语义的核心是在数据的相互关系中定义数据的含义。

1.2.2 语义信息模型的用途

针对不同目的，语义信息模型可以采用多种形式以及抽象层次。语义信息模型中包含的信息量要满足以下几种目的：

(1) 问题域基本结构的抽象说明。这种抽象的说明包括领域中最通用的信息及有关知识，以使各方面的利益相关者能够相互理解，并达成对问题域认识的一致。

(2) 全面把握复杂问题域的有关信息。语义信息模型是对问题域有关信息的查找、过滤、检查、重获、组织、重获以及抽象。通过模型研究系统内各组成部分之间的依赖关系，全面把握问题域的信息流。

(3) 对问题域全面或部分的描述。问题域作为一个独立系统不需要进行分解的情况在实际中是很少见的。较通常的情况是，问题域可以分解为相互区别、不连续的描述单元。这些单元作为整体描述的一部分被单独存储和操纵。其中的某些单元由于在不同的抽象层次上具有相似特征，而被认为是同一种类型。语义信息模型需要对反映认识结果的种类进行描述。同时模型要能够描述问题域中典型的事实范例。

从而能够指导信息系统的数据建模，系统的数据处理的设计与实现。

1.2.3 语义信息模型的内容

语义信息模型和任何其他模型一样也包括语法、语义和语境。语法和语境都是为模型所要表达的语义服务的。在具体的工程实践中，通常要根据信息模型所要表达的语义，选择或开发相应的语法规则。由于信息模型要描述数据资源以及数据资源之间的关系，语义信息模型中的语法一般要具有类、关联、状态、实例、规则和消息等语义模型元素。

语义信息模型自身是一个为了在计算机系统中处理、加工、利用信息资源而开发的人工制品，被应用于一个提供模型含义的大型语境中。这个语境包括模型的内部结构、模型建立时的假设、模型应用时的前提条件、模型分析、建立过程的依据和说明、缺省值集合以及模型与其所处环境之间的关系。

1.2.4 语义信息模型的开发

语义信息模型的开发是信息系统开发的一部分，现有的语义数据建模方法都是在从不同的角度努力构造着系统涉及问题域的语义信息模型。语义信息模型的开发方法和工具随着需求变化和时代发展不断地前进。

语义数据模型发展的最初动力是为了克服传统数据模型的缺陷，提供不受具体实现结构限制的方法，即与数据处理的物理实现无关和更多地面向用户的模型。语义数据模型提供了一种"自然"的机制来说明数据库的设计，同时更准确地表示数据及其之间的关系。

语义数据模型主要包括实体关系模型(Entity Relationship Model，简称 E-R 模型)、RM/T 模型、TAXIS 模型、SDM 模型、函数模型、SAM 模型以及 SHM+模型等。其中，以实体关系模型为代表。E-R 模型经过多年的发展、完善，逐渐形成了完整、统一的建模标准，同时有许多软件产品支持用 E-R 模型完成数据库的概念、逻辑和物理建模过程。

前面提到过的 IDEF 方法家族是这一方面应用最为广泛的开发方法。由于问题本身的复杂性，一种表达形式一般只可能适用于一类特征。IDEF 家族发展了各种不同的方法来描述不同的特征。其中的 IDEF5 被称为本体论描述获取(Ontology Description Capture)，是一种正在逐渐兴起的语义信息建模方法。这种建模方法还不是很成熟，国内也没有任何机构应用它开发过大型的项目。

从知识共享的角度看，本体论可以被看做是一种概念化的显式说明或表示，是对客观存在的概念和关系的描述。从表面上看，本体论对问题域的描述与数据字典没有太大的区别，也是

许多名词的集合。人们经过进一步研究后,发现两者存在着很大的差距。本体论采用的文法和公理一般采用精确的形式语言、精确的句法和明确定义的语义。本体论的方法并不仅局限于此,这种方法努力使问题域中的概念与概念、概念与对象、对象与对象之间的关系以及在问题域中对象上所施加的约束明确定义,而不是隐含在分析者的头脑中或实现者的程序中,从而大大减小对问题域中概念和逻辑关系可能造成的误解。本书中将对这一方法进行探讨。

1.3 XML与语义信息模型

许多文献都称XML是语义自描述的,是一种语义信息模型。那么XML与语义信息模型究竟是什么关系呢?这一节我们分析一下这个问题。

1.3.1 Web与语义信息模型

Web应用已经扩展到了人类活动的任何一个方面,远远超出面向文档系统的范围,尽管Web一开始被想像成一个文档系统。对于Web应用的巨大需求,包括全新的和源于对现有系统改造的需求,加上缺乏熟练的IT人才,导致了对更好的软件工程方法的强烈要求。这种情况有点类似于在软件领域采用的较成熟方法的发展,如数据库技术和面向对象方法的发展过程。

电子商务、数字图书馆和远程学习等Internet领域的应用具有多项特征混合在一起的特点。这一特点使得它们与以前的信息技术有着根本的不同:

(1) 对于具有有限计算机应用经验或是没有计算机应用经验的用户,通用存取技术需要一个新的人机界面,可以捕获用户的注意力,简化信息存储。

(2) 不同种类的信息资源的全球有效性需要对可能存储于不同系统(数据库、文件系统、多媒体存储设备)和分布存储于多个站点的结构化和非结构化信息进行一体化管理。

近几年,WWW已被推选为开发Internet应用的理想平台,这归功于它强大的基于多媒体性能和浏览方法的信息交流样式,还要归功于它开放的构造标准。这些都促进了不同类型内容和系统的一体化。

现代Web应用通常被描述为超媒体和信息系统的混合。由于其混合特性,Web应用的开发面临着一些实用性的要求:

(1) 需要处理结构化数据(例如数据库记录)和非结构化数据(例如多媒体数据)。

(2) 通过导航界面支持探索性的数据访问。

(3) 高水平的图形质量。

(4) 内容结构的用户化和可能的动态适应性,导航原语和表现风格。

(5) 支持前摄行为,例如,推荐和过滤。

为了提高效率,扩大网站开发所面临任务的覆盖度势在必行。因为市场上绝大多数的网站开发工具只是集中于设计和实现,而不注意需求分析和概念建模,因此,实现并维护一个大的网站是一项人力密集、易出差错的活动,这种活动很难从采用模型概念和CASE工具的规范开发过程中获益。

为了解决这些问题,研究人员提出了所谓的模型驱动网站设计方法。这类方法采用了以半结构化符号表达数据结构和网站的超文本拓扑的思想,同时使用概念级规范来驱动设计和实现的思想。

在这类 Web 开发方法中，完成需求分析之后，有一个被称为概念化(Conceptualization)的设计阶段。在这一阶段中应用被一组覆盖预期解决方案主要组成部分的抽象模型所表示。在 Web 环境下，概念化与信息系统设计中的类似行为有着显著的风格差异。在 Web 环境下的概念化将主要精力集中于获取展现给用户的对象和对象之间的关系，而不是这些对象和对象之间的关系如何在软件系统中表示。尽管它们使用的符号可能是相同的，但是由此概念化产生的 Web 应用视图却有别于一般的数据库应用。

1.3.2 XML 与语义信息模型

XML 对统一结构化语法和半结构化语法的承诺，又重新燃起了人们将一些几乎不可能的事变成切实可行的希望。那么，什么是 XML 的语义？因为语义这个词的特殊性，每个人对语义定义的观点都各有不同。一般来说，语义是构建在公用语法上的系统中 XML 数据的一层规范。按照 Uche Ogbuji 的想法，标记 XML 语义的概念如下，当然，在这三概念之间有一些重叠：

(1) 元素类型名称、属性名称和某些情况下内容术语的解释。

(2) 用于用有效文档引导事务的处理规则(也称作商业规则)。

(3) 一个文档中的结构化元素与另一个文档中的结构化元素之间的关系。

当我们提到 XML 时，通常并不单指 XML 协议本身，更普遍的情况是指以 XML 为基础的相关协议簇以及由此而带来解决问题的新方法。就 XML 本身而言，XML 目前有明确的语法和结构，但它没有提语义透明性。语义透明性可以使 XML 机器建立元素(比如，PurchaseOrder 或 PO)和根据该元素执行专门操作的高阶处理之间的关系。总而言之，它意味着数据中的表达式如实地表示了相应概念的含义。语义透明性的最终测试是如果某个人只使用适用于 XML 处理软件的机制，它能否正确理解 XML 数据的含义。显然，单靠 XML 根本无法实现语义透明性，这正是众多 XML 技术专家关注语义透明性的原因。

那么为什么人们会对 XML 的出现而欢欣鼓舞并寄予厚望？原因是多方面的，其中最重要的就是：XML 是一种完全面向数据语义的标志语言，取消了 HTML 的显示样式与布局描述能力，突出了数据的语义与元素结构描述能力。具体地讲包括以下几个方面：

(1) XML 使用 ASCII 编码。

(2) 以内容为中心，符合 Web 的主旨。

(3) 构建于 Web 标准和概念之上。

(4) 沿袭 SGML 的传统。

(5) 与 EDI 兼容。

(6) 被广泛采纳和使用。

(7) 应用领域广泛。

我们将在本书中以 XML 为具体手段，以 Web 应用为背景介绍语义信息模型的多种应用，探讨 XML 在语义信息模型中的应用方法。下面首先介绍 XML 与数据的语义表示，作为进一步研究的基础。

1.4 XML 与数据的语义表示

早期的计算机使用的数据和处理它的程序都固化在穿孔卡片上，数据的语义和数据的处

理合二为一。后来，数据与它的部分语义被抽取出来，放到特殊格式的数据文件里，如 dbf 文件。文件头部包含数据的部分语义，另一部分语义依然保留在处理它的程序里。在这个发展过程中，数据的语义与处理它的程序的耦合程度由紧密走向分离。分离意味着灵活性和共享性。这一发展方向预示着数据的语义最终会独立出来，数据不再只是数据，而是带有语义的信息；处理数据的程序不再只是专有的、惟一的，而是共享的、开放的。在开放的环境中，软件的互操作才真正成为可能，这是一条"标准化"的道路。

W3C(World Wide Web Consortium)开发的 XML(eXtensible Markup Language，可扩展标记语言)正是实现这一目标的产物。W3C 创立于 1994 年 10 月，致力于领导万维网(World Wide Web)，制定公共的协议，促进万维网的发展并确保其互操作性。W3C 在世界各地已有 400 多个成员组织，并在推进万维网的发展方面赢得了广泛的国际赞誉。

1.4.1 数据表示的第一次统一

什么是数制？用一组固定的数字和一套统一的规则来表示数目的方法就叫做数制(Number System)。人们比较熟悉的是十进制，即 0,1,2,3,4,5,6,7,8,9 这十个数。在计算机领域中，还有另外 3 种数制：二进制、八进制和十六进制。八进制的基数是 8，十六进制的基数是 16。十六进制用 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F 来表示，这些数制原理与十进制原理相似。二进制是计算机与网络通信中采用的基本数制，而八进制和十六进制则是二进制的压缩形式。这些数制之间是可以相互转换的。

什么是数据？数据指人们看到的形象和听到的事实。自从有了计算机，就存在两种形态的数据。一种形态称为人类可读形式的数据，简称人读数据，是人类使用的语言、文字、数字以及图像。另一种形态称为机器可读形式的数据，简称机读数据，是计算机能够识别的二进制数的形式。在二进制中，数据的最小单位就是二进制的一位数(0 或 1)，简称为位，英文名称是比特(bit)。正是通过这些 0 和 1 的组合，可以将人读数据中的所有基本符号——字母、数字以及专门符号表示出来。在计算机中，8 位为一个字节(Byte)，这是计算机进行数据处理和存储的基本单位。此外，我们还会遇到另一种单位：机器字长，表示计算机一次能够处理数据的位数，是衡量机器能力强弱的标志之一。最初的微型机只有 8 位，后来发展到 16 位机、32 位机，直到字长为 64 位的巨型机。

什么是编码？编码就是用二进制表示字母、数字以及专门符号的一串数字。这是一个涉及世界范围内有关信息表示、交换、处理和存储的基本问题，一般都以国家标准或国际标准的形式颁布施行。目前国际上通用的编码系统为美国标准信息交换码(ASCII)。为了适应汉字信息交换的需要，我国于 1981 年颁布了汉字编码系统，这个系统包含 6763 个常用汉字，在国际上也适用。有了这两套编码系统，人们才真正可以用汉语与计算机"对话"。

通过编码把人读数据转化为机读数据的过程是由计算机硬件系统和软件系统完成的。硬件系统只能识别 0 和 1，软件系统分为计算机本身运行所需的系统软件和用户完成特定任务所需的应用软件。

任何文字、图形等数据，只要用计算机处理就变成了 0 和 1，即所谓数字化，这一进步把人类带入了新的时代。任何信息只要通过计算机处理就进入了新的信息管理状态，其共享、传递和储存都有了新的方式。正是在这个意义上，人们开始了信息编码化的历程。

在 Licklider 提出"电脑与人类交流"的思想之后，1963 年，一位在电脑发展史上做出重大贡献的人物终于制定出统一的信息表示方法，即 ASCII(美国信息交换标准码)编码。这为

Licklider 思想的实施,在技术层面上给予了强有力的帮助。这位伟大的人物就是后来被尊称为"ASCII 之父"的 Bob Bemer。

最初,ASCII 是由 128 个由数字 0 和 1 组成的 7 位二进制字符串构成的。每一个字串代表了英文字母表中的一个字母、阿拉伯数字、标点符号和一些特定的符号。人们现在使用的电子邮件、World Wide Web、激光打印机和光盘游戏都应该归功于这项技术的突破。回头看看 ASCII 出现之前的计算机构造,你会觉得 ASCII 的出现竟是如此的重要。

在 ASCII 出现之前,不同的计算机之间无法相互通信。每家制造商都使用自己的方式来表示字母、数字和控制码。那时,在计算机中表示字符的方式就有 60 多种。IBM 的设备中就使用了 9 种不同的字符集。电脑之间的相互对话都无法完成,更别说与外界对话了。

今天,古老的 ASCII 作为一种字符集标准,已被广泛应用于计算机设备和大多数操作系统中。可实际上,自 1963 年 ASCII 编写完毕到它被普遍采用总共花费了 18 年的时间。这与 IBM 及其 System/360 系统有关。当 ASCII 正在开发之际,每个人,甚至包括 IBM 的人在内,都认为该公司会采用这种新标准。在此之前,IBM 使用穿孔卡代码的扩展码 EBCDIC。但是,正当 ASCII 完成和 System/360 准备推出时,IBM 的 OS/360 开发小组组长 Frederick Brook 告诉 Bemer,穿孔卡和打印机还没为 ASCII 做好准备。这时 IBM 只好为 System/360 开发一种在 ASCII 和 EBCDIC 之间转换的方式。可惜最后开发的技术却未能奏效。

直到 1981 年,IBM 最终开始在 PC 中使用 ASCII 标准。至此 ASCII 才真正成为计算机交流的标准。

ASCII 虽然诞生于 1963 年,但至今仍保持活力。虽然在一些新型的操作系统中使用了另一套的编码方案,如 Windows NT,但它们都必须与 ASCII 保持兼容。ASCII 的出现,使得电脑信息表示达成统一,为以后电脑的联网交流奠定了基础。

1.4.2 XML 的语义数据表示

XML 提供了一个将大量信息组织成为具有确定意义的整体结构的功能,这种具有确定意义的整体称为文档。XML 文档是数据的容器,它并不关心数据的显示样式与布局效果。构造 XML 文档的基本成分是元素(Element),元素由标记(Tag)定义,包括开始标记和结束标记。XML 文档的元素形成一种层次结构,处于顶端的元素称为根元素,根元素包含了所有其他元素。XML 支持元素的嵌套,但不像 HTML 那样允许元素交迭,即 XML 中元素结束标记出现次序应该和开始标记出现次序严格相反。XML 通过元素的这种层次关系支持丰富数据结构(Rich Data Structure)。换句话说,XML 文档中的元素之间不是简单的前后次序关系,而是具有明确的从属、依赖等关系。

下面让我们来看一个简单的例子。

程序清单 1.1

```
<? xml version='1.0'? >
<books>
    <book category='reference'>
        <author>Nigel Rees</author>
        <title>Sayings of the Century</title>
        <price>8.95</price>
    </book>
```

```
        <book category='fiction'>
            <author>Evelyn Waugh</author>
            <title>Sword of Honour</title>
            <price>12.99</price>
        </book>
    </books>
```

结构良好(Well-formed)是XML中一个重要的概念,在XML中它不是指代码排列整齐美观之类,而是具有明确的定义。一个结构良好的文档,其标记是可预测的,普通XML解析器也能读取它。XML解析器的主要任务是验证文档的结构,并为访问文档提供一个编程界面。也有一些专用解析器能够验证文档的内容,称为验证解析器。当前,我们已经可以得到大多数平台(包括Windows、UNIX、OS 400、MVS)上的XML解析器。这些解析器主要有两种形式,即面向平台的解析器和跨平台的Java解析器。例如IBM的XML4J(XML for Java)解析器,通过它我们可以在编程环境中访问文档的DOM模型(Document Object Model),文档中的各个元素成为树状结构中的节点对象,可以方便地进行遍历、增加节点、删除节点和修改节点等操作。

XML和HTML是两个独立但并不对立的标准。HTML的目的在于标示数据以便于在浏览器中显示,而XML的目的在于标示数据以便机器处理,特别强调数据的语义与元素之间的关系。对计算机来说,从XML文档中提取数据比从HTML文档来得容易。因此,当Web上出现更多的XML内容时,Web搜索的精确程度也必定会有很大的提高。HTML的标记数量是固定的,但XML允许用户自己定义元素。那么,当每个用户都定义自己的标记时他们又如何协作呢?这就是XML所面临的亟待解决的问题,同时也是这种新技术所带来的机遇。XML只有在合作伙伴以至竞争对手能够共同协作,定义本行业通用的"词汇表"时才能发挥出最大的效益。XML词汇表定义的是特定行业的一组标准元素,并描述这些元素如何适应文档的整体结构。例如,在医疗与健康行业,HL7标准化组织正在为临床数据开发一个通用的XML词汇表。使用这个词汇表,不同的医生可以方便地共享信息。又如,在金融和零售业,XML标准也正在浮现。Microsoft和Intuit与金融机构协作为联机银行业务开发了一个开放金融交易(Open Financial Exchange,OFX)XML词汇表。信用卡公司和零售商合作开发了开放贸易协议(Open Trading Protocol,OTP),这是一种面向支付、报价等业务的可协同操作的消息协议。无论是OFX还是OTP,都说明了竞争者可以从这种竞争中获得好处。因此,新词汇表的定义引起了人们极大的兴趣。那些在本行业词汇表定义中处于领导地位的公司将有更多的机会使新词汇表符合自己的商务语言。除了定义词汇表之外,所有的公司都面临着应用新词汇表的问题。也许在不久的将来可能出现以提供XML入门或词汇表来解释这类服务赢利的企业。

可以想像,许多文档将包含多个词汇表。在一个病历文档中可能包含了病人词汇表和医生词汇表,这两个词汇表分别用Address元素标示病人和医生的地址信息,因此,程序在寻找病人住址时应该将它与医生住址分开。如果Address元素含义不明确,搜索病人地址的程序就可能使用错误的元素。XML的命名空间(Name Space)就是上面这种引用多个词汇表时出现的冲突的解决方法。命名空间通过统一资源标识符(Uniform Resource Identifier,URI)限定元素名字,从而避免了冲突。

为支持词汇表,XML提供了一种定义文档结构与元素的机制,这就是文档类型定义(Doc-

ument Type Definition,DTD)。DTD既可以作为一人独立的文档提供,也可以嵌入到 XML 文档内。当解析器接收到一个附带 DTD 的 XML 文档时,它就会检查文档是否结构良好,是否符合 DTD 中的类型定义。如果 XML 文档能够通过上述测试,它就是“合法的”。但是,DTD 也有不足之处,如不能定义元素的数据类型。鉴于 DTD 的这种限制,一些厂商提交了新的 XML 模式(数据定义)建议,如 Microsoft、IBM、Textuality 向 W3C 提交的文档内容定义(Document Content Definition,DCD)建议。DCD 提供了定义文档结构、约束及附加属性(如基本数据类型)的方法。标准 SQL 和流行的编程语言都支持 DCD 建议中的数据类型。

人们称 XML 为“Web 上的 ASCII 码”。从 ASCII 的发展历史可以看出,信息的表示向标准化每走一步,整个信息技术也向前发展一步,与 ASCII 对整个信息技术的促进作用相比,基于语义的信息表示(XML)的意义将更大,一方面它同 ASCII 一样继承了标准化的思想,统一了数据表示的“语法”,另一方面,它的目标更加远大,它在朝着统一“语义”的方向迈进。

1.5 XML 的应用进展

本节主要探讨 XML 的应用进展,下面内容可以看成本书大部分内容的概述,希望读者对全书有一个大致的了解。

1. RDF 及其应用

如果说 XML 是一种语言的能力,那么 XML 应用程序就是特殊的语言。资源描述框架(RDF)就是这样的一个 XML 应用程序:一个使用 XML 语法的数据-模型语言。RDF 是一种描述和使用数据的方法,这就是说 RDF 是关于数据的数据,或者说是元数据。

RDF 为 Web 资源描述提供了一种通用框架,它以一种机器可理解的方式来表示,可以很方便地进行数据交换。RDF 提供了 Web 数据集成的元数据解决方案。通过 RDF 的帮助,Web 可以实现目前还很难实现的一系列应用,可以更有效地发现资源,提供个性化服务,分级与过滤 Web 的内容,建立信任机制,实现智能浏览和语义 Web 等。第 2 章将专门介绍 RDF 及其应用。

2. XML 的语义表示和处理方法

XML 作为一种半结构化的数据,其特点是可以携带自描述信息,但是如果“H1”和“HEADER”作为自描述信息时没有进一步说明,那么机器只能将它们当成字符串处理。为了充分利用 XML 的自描述信息和概念化领域模型,有必要在本体和 XML 的结构之间建立一种对应的关系。因此,XML 的语义表示和处理方法,必须把 XML 与本体(Ontology)技术相结合。

在第 3 章中,首先从语言的角度对 XML 在语义表示方面的优势和不足进行分析,归纳出 XML 在语义互操作方面的需求,从而引入本体的概念。在此基础上,深入分析并总结本体的构造方法论及基于本体构造领域信息语义模型的理论,并结合 XML 标准介绍当前具有代表意义的本体表示和交换语言 XOL 和 OIL,同时对 RDFS 用于本体表示的方法进行探讨,最后介绍一种将描述逻辑表示的本体转化为 DTD 的方法。

3. XML 与语义 Web

语义 Web 是 Tim Berners-Lee 以及 Web 和 W3C 的始创者提出的称做“Semantic Web”的新一代 Web,其基本思想就是通过在 Web 信息的创作和发布中嵌入机器可阅读的、代表某类知识的标注,使 Web 上的数据不仅能够被机器用于显示,而且能够被机器所理解,从而能提高

信息服务的质量,并开拓各种崭新的、智能化的信息服务。如果进一步将这些体现了数据与应用之间联系的知识以对用户透明的方式嵌入各种不同的信息源,则 Web 页面、数据库、程序、模块和探测设备将通过能够处理这种信息表示方法的 agent 连结起来,相互之间能够理解和协作。如果能够成为现实,那么这种方法将产生革命性的力量,并且改变人类与信息交互的方式。

4. XML 在人工智能中的应用

人工智能是计算机科学的一个分支,其长远目标是人工智能的机器实现。经过半个多世纪的发展,人工智能"技术"产生了一大批理论成果,并获得了广泛的应用。

AI 和其他领域的研究者分别从"学习"的角度和信息表示的角度研究语义 Web。在研究过程中,人们逐渐认识到知识表示在使 Web 更加"智能"的过程中的基础地位。

XML 技术为在因特网环境下的应用奠定了基础,使人工智能的各个方面,特别是知识工程方面的研究成果能够在 Web 环境中广泛应用。基于 XML 的标记语言能够为分布于网上的知识提供一种统一的存储(Storage)和交换(Interchange)格式,真正实现因特网上知识交互、重用和共享。将 XML 技术运用于知识的表示出现了一门新的技术——知识标记技术(Knowledge Markup Techniques),并且已经成为国外大学的一门课程。我们将在第 5 章介绍 XML 在人工智能中的应用。

5. XML 与分布计算技术

支持 XML 的 Web 应用程序能够连接起来形成系统。到目前为止,大多数编程者都把 Web 应用程序严格地视做客户机/服务器结构,Web 客户机从 Web 服务器上获取信息,当一个服务器访问数据库时,它并不会向另一台 Web 服务器寻求帮助。随着时间的流逝,越来越多的应用程序资源开始由 Web 服务器控制,实现多个服务器共同解决问题的能力就变得更加重要。由于一台服务器能够呼叫另一台服务器以寻求数据和处理能力,因而可以在现有应用的基础上编制出精致的分布式程序。既然这些系统经常使用不同的服务器软件和分布式计算技术,所以就需要 XML 来提供一个抽象层来集成不同的系统。从另一个服务器获取 XML,进行操作,然后把结果传递给客户机可以满足客户机请求的需要。许多使用 XML 来有效地完成这类工作的技术正处在开发当中,其中包括 XML-RPC、简单对象访问协议(Simple Object Access Protocol,SOAP)以及分布式 Web 数据交换(Web Distributed Data Exchange,WDDX)。

XML-RPC 是一种远程执行驻留在服务器上的进程的协议。这与传统的 RPC 非常相似,后者允许命名一个过程以便执行并可以提供一个参数列表。XML-RPC 将 XML 当做完成这类工作的途径,这样能够减少与特定平台有关的问题。对于编程人员来说,XML 使本地资源可用于远程执行问题简单化了。XML-RPC 已经在许多常见平台上得以实现。

SOAP 与 XML-RPC 类似,也使用 XML 来访问 HTTP 之上的对象的方法和属性。XML 用来描述被调用的方法和被传递的数据,这样能够避免对任何特定类型的分布式对象技术的依赖。

WDDX 是一种使用 XML 串行化数据结构的技术。例如,它可以用做通过 Internet 返回数据库结果的低级机制。

在第 6 章中我们将介绍 XML 在分布计算技术中的应用。希望所提供的知识能够有助于更好地在基于 Web 的资源的基础上构建多层分布式系统。

6. XML 与关系数据库

用 Web 前端连接关系型数据库是相当常见的。但是 XML 的数据模式天生就是层次结构

的，这会使得在将它们和大多数普通数据库使用的关系型模式相匹配时遇到一些困难。虽然使用 XML 作为关系型数据库的接口并不直接利用 XML 的独特功能，但它能将现有的数据引入到新的系统中。既然 XML 是一种流行的，并且平台无关的连接方式，那么编程者肯定需要一条途径作为 XML 和数据库之间的接口。许多数据库厂商在意识到这一事实后，已经开始在自己的引擎中增加对 XML 的本机支持。

第 7 章中将探讨 XML 词汇表映射到关系型表以及相反操作时的有效策略，了解 XML 能够从哪些层次去改变应用程序与数据结构的接口方式。

7. XML 与 Web 服务

Web 服务是下一代的 WWW，它允许在 Web 站点上放置可编程的元素，使之能进行基于 Web 的分布式计算和处理。

微软公司雄心勃勃的.Net 战略核心概念就是"把软件当做服务"，也就是把软件应用产品与商业、内容和信息服务合并成一种事物，使之成为可以在网络上订阅使用的服务形式。人们设计、构造、实施、运作、集成和使用软件的方式都将透过网络完成。"总有一天，所有的应用软件将被设计成一种服务，可以在网上订购。"这是微软一句富有感召力的口号。

Sun 公司也提出了概念相似的开放式网络环境(Open Net Environment，ONE)战略。Oracle 公司则以 Dynamic Web Service 的概念表示与微软抗衡。这些概念的技术取向都是大同小异的，就是都将 XML 作为支持 Web 服务以及各种控制过程的技术核心。第 8 章介绍这方面的内容。

第2章　资源描述框架(RDF)

资源描述框架(RDF)不仅是一个简单的元数据方案,而且是一个能对结构化的元数据进行编码、交换及再利用的体系框架。这种体系结构通过对通常意义上的语义、语法和结构的支持,提供了在各种不同的元数据体系之间的互操作性。本章详细说明RDF模型和语法规范以及RDF模式规范(RDF Schema),介绍基于RDF的Web数据集成方案和Web服务集成方案。

2.1　概述

如果说XML提供表达一种语言的能力,那么XML应用程序就是采用XML表达的一种特殊语言。资源描述框架(RDF)就是这样的一个XML应用程序:一个使用XML语法的数据-模型语言。RDF是一种描述和使用数据的方法,也就是说RDF是关于数据的数据,或者说是元数据。

在当今的社会中,信息无处不在,从这些信息中获取对自己有用的信息并不是件容易的事。当然也有例外,比如在图书馆里可以根据书名、作者名或关键字的信息找到藏书号,从而很容易找到所要的书,在音像店里可以根据片名、主演等信息方便地找到自己所要的影碟。这两个系统有一个共同的特点——它们都是建立在元数据之上。

元数据是关于数据的数据或关于信息的信息。例如:书的文本就是书的数据,而书名、作者、版权数据都是书的元数据。元数据并不一定就是用来检索的,也可用于内部的管理,如图书馆系统可以为书定义被借次数这个元数据,以了解书的被借阅情况,确定是否要增加副本数。元数据的使用,可以大大提高系统的检索和管理的效率。

网络是个大的数据库,它里面包含的数据比起图书馆和音像店来要复杂得多,五花八门,什么都有,但有一个问题——网络基本上没有元数据。那搜索引擎是怎么工作的呢?其实搜索引擎中除极少数如Yahoo!外,基本上都是采用网页的全文检索来提供检索服务,这就可想而知其查准率之低。Yahoo!将其收集到的网站及网页分门别类加以索引和文摘(由人工完成),从而大大提高了查准率,这也是其流行的一个重要原因。但对浩瀚的信息海洋若都采用人工标引显然是不现实的,所以用Yahoo!检索的时候查全率不如像AltaVista、Infoseek这样的搜索引擎高,原因就是其收录的网站网页数量有限。如果网络上的资源在创建之初就都使用元数据来描述其自身的信息,那么就可以省去人工标引的麻烦。但是怎样用元数据来描述,这得有个标准,W3C提出的用于描述Web资源的RDF(Resource Description Framework资源描述框架)就是这样的一个标准,RDF给出了Web数据集成的元数据解决方案。

资源描述框架(RDF)是处理元数据的基础,它提供了Web上应用程序间交换机器能理解的信息的互操作性。RDF使用XML来交换Web资源的描述,但被描述的资源可以是任何类型,包括XML资源和非XML资源。RDF强调Web资源的自动处理功能。RDF能被用于广泛的应用领域,如在资源发现中提供更好的查询引擎能力,在特定站点、页面和数字书库上描述内容和内容间关系的编目,通过智能软件主体实现知识共享和交换,内容评估,描述代表单个逻辑"文档"的页面聚集,描述页面的智能属性以及表达站点的私有政策,同样也表达用户的

私有偏好。对电子贸易、合作和其他应用，带有数字签名的 RDF 将成为建立“可信任 Web”的关键。

RDF 的含义就是描述资源的框架(Framework for Describing Resources)，下面逐个来看这三个词的意思。

资源(Resource)：所有在 Web 上被命名、具有 URI(Unified Resource Identifier，统一资源标志符)的东西。如网页、XML 文档中的元素等；

描述(Decription)：对资源属性(Property)的一个声明(Statement)，以表明资源的特性或者资源之间的联系；

框架(Framework)：与被描述资源无关的通用模型，以包容和管理资源的多样性、不一致性和重复性。

综合起来，RDF 就是定义了一种通用的框架，即资源—属性—值的三元组，以不变应万变，来描述 Web 上的各种资源。

下面我们来看一个简单的 RDF 的例子：

<rdf : Description about = 'http://www.textuality.com/RDF/Why-RDF.html'>（指明被描述资源的 URI）

<Author> Tim Bray </Author>（被描述资源有一个 Author 属性，其值是 Tim Bray）

<Home-Page rdf:resource = 'http://www.textuality.com/'/>（被描述资源有一叫 Home-Page 属性，其值指向另一资源）

</rdf: Description>（结束标志）

RDF 实际上包含了两个正在形成中的新标准——RDF Model and Syntax([RDFMS]，RDF 模型和语法规范)和 RDF Schema(RDF 模式规范)。RDF 提供了一种基本的结构，用于在 Web 上对元数据进行编码、交换和重用。

资源描述框架(RDF)模型和语法规范介绍了一个模型来表示 RDF 元数据，同时介绍一种以某种方式编码和传输这种元数据的语法。这种方式最大限度地利用了独立开发的 Web 服务器和客户协同工作的能力。语法使用了扩展标记语言 XML，RDF 的目标之一就是以一种标准的、共同使用的方式规范 XML 基础之上的数据的语义。RDF 和 XML 是相互补充的，RDF 是一个元数据的模型，只通过引用涉及许多传输和文件存储所需的编码问题(如国际化，字符集等)。对于这些问题，RDF 依赖着 XML 的支持。同样 XML 语法对 RDF 而言只是一种可选的语法，可能会出现表示相同 RDF 数据模型的替代方法，明白这一点也是很重要的。

Web 应用使用的描述可被建模成 Web 资源间的关系。RDF 数据模型，如[RDFMS]中所说明的，就命名属性和值而言，定义了描述资源间相互关系的一种简单的模型。RDF 属性可被认为是资源的属性，对应于传统的属性与其值的配对。RDF 属性同样代表了资源间的关系。因此，RDF 数据模型就和实体－关系图类似。但是 RDF 数据模型没有提供机制来说明这些属性，也没有提供任何机制来定义这些属性和其他资源间的关系。那是 RDF 模式的任务。

模式规范进一步来完善此框架。为方便元数据的定义，RDF 将具有一个和许多面向对象编程和模型系统相似的类系统。一个类的集合(是为一个特定目的或领域而建的)称为一个模式。类被组织成一个层次，通过子类的改善而提供了可伸展性。使用这种方法，当创建一个和已存在的模式略有不同的模式时，只需在基本的模式上进行增加和修改。通过模式的共享，RDF 将支持元数据定义的重用。由于 RDF 增加的可扩展性，处理元数据的主体将能够从它

们不熟悉的模式回溯到已知的模式,并在元数据上执行其原本没有设计去处理的有意义的动作。RDF 的可共享和可扩展性同样允许元数据的作者使用多重继承来“混合”定义,提供其对数据的多个观点或影响其他人所做的工作。另外,还有可能在来自多个来源(如不同元数据类型“交叉”)的多个模式基础上创建 RDF 实例。

RDF 模式机制提供了一个应用于 RDF 模型中的基本类型系统。它定义了资源和属性如 rdfs: Class 和 rdfs: subClassOf,其使用于特定应用——特定模式。

RDF 模式类型系统和面向对象的编程语言如 Java 的类型系统相似。但是 RDF 和这类系统不同之处在于不是就其实例可能具有的属性定义一个类,而是就它们所应用的资源的类定义属性。这是 rdf:domain 和 rdf:range 约束的任务。比如可定义 author 属性具有领域 Book 和范围 Literal,而一个经典的面向对象系统可能典型地定义一个类 Book 带有一个称为 author 的属性和 Literal 的类型。RDF 以属性为中心的方法的一个好处是任何人都可以就现存的资源说一些他们想说的事情,这是 Web 的结构原则之一。

RDF 的广泛的目标是为描述资源定义一种机制,这种机制没有为特定的应用领域做任何假设,也没有为任何应用领域(先验地)定义语义学。机制的定义应当是领域中立的,但是机制应当适合于描述任何领域的信息。

2.2 RDF 模型和语法

资源描述框架(RDF)模型和语法规范介绍一个模型来表示 RDF 元数据,同时介绍一种以某种方式编码和传输这种元数据的语法,这种方式最大限度地利用了独立开发的 Web 服务器和客户协同工作的能力。

2.2.1 基本 RDF

2.2.1.1 基本的 RDF 模型

RDF 的基础是表示命名属性和属性值的模型。RDF 模型运用来自各个数据表示组织的已为大众所接受的原则。RDF 属性可被认为是资源的属性,对应于传统的属性与值的组合对。RDF 属性也表示资源和 RDF 模型间的关系,故能组装成实体-关系图(更准确地讲 RDF 模式——其本身是 RDF 数据模型的实例——就是实体-关系图)。在面向对象的设计术语中,资源对应于对象,属性对应于实例变量。

RDF 数据模型是一种语法中立的表达 RDF 表示式的方法。两个 RDF 表示式是相等的,当且仅当它们的数据模式表达式是相同的。这个相等的定义允许一些语法上的变化而不改变含意。

基本的 RDF 数据模型由三种对象类型组成:

资源

由 RDF 表示式描述的所有东西都可称为资源。资源可以是整个的 Web 页面,比如 HTML 文档“http://www.w3.org/Overview.html”;也可以是一个 Web 页面的一部分,比如一个文档内的特定 HTML 和 XML 元素;还可以是多个页面的集合,比如整个 Web 站点。资源还可以是不能通过 Web 直接访问的对象,比如一本印刷的书。资源通常由 URI 加锚定符命名。任何事物都可具有一 URI,URI 的扩展允许了任何可以想像的实体的引入。

属性

属性用于描述一资源的特定方面、特征、属性和关系。每个属性具有一特定的含意,定义其允许值、可描述的资源类型、其他属性的关系。

声明

一个特定的资源加上该资源一命名的属性及属性的值构成一个 RDF 声明。声明的这三个独立部分分别称为主体、谓词和客体。声明的客体(即属性值)可以是另一个资源或文字,也就是说,由 URI 指定的资源或者一个简单的字符串以及另一个由 XML 定义的简单数据类型。

下面一个句子是一个简单的例子:

Ora Lassila is the creator of the resource http://www.w3.org/Home/Lassila.

这个句子具有如下的部分:

主体(资源)	http://www.w3.org/Home/Lassila
谓词(属性)	Creator
客体(文字)	"Ora Lassila"

这个句子可由图 2.1 表示:节点(椭圆)表示资源,弧表示命名的属性,矩形结点表示字符串文字。

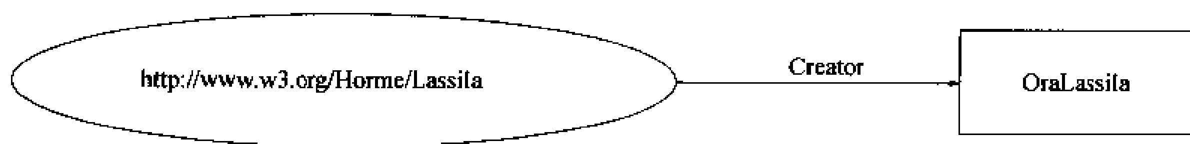


图 2.1 简单的结点和弧图

2.2.1.2 基本的 RDF 语法

RDF 数据模型提供了一个抽象的概念化的框架来定义和使用元数据。同时还需要一个具体的语法来创建和交换这一元数据。RDF 的模型和语法规范使用扩展标记语言 XML 编码作为其交换语法。RDF 同样需要 XML 名字空间(Namespace)功能来精确地将每一个属性和定义属性的模式联系起来。

基本序列化(Serialization)语法

RDF XML 语法将同一资源的多个声明并组到一个描述元素(Description)中。描述元素在一个 about 属性中命名资源。如果资源不存在(即还没有资源标志符),则描述元素使用一个 ID 属性来为每个资源提供一个标志符。

基本 RDF 序列化语法具有如下的形式:

```
[1] RDF            ::= ['<rdf:RDF>'] description * ['</rdf:RDF>']
[2] description    ::= '<rdf:Description' idAboutAttr? '>' propertyElt *
                       '</rdf:Description>'
[3] idAboutAttr    ::= idAttr | aboutAttr
[4] aboutAttr      ::= 'about="' URI-reference '"'
[5] idAttr         ::= 'ID="' IDsymbol '"'
[6] propertyElt    ::= '<' propName '>' value '</' propName '>'
```

```
                    | '<' propName resourceAttr '/>'
  [7] propName      ::= Qname
  [8] value         ::= description | string
  [9] resourceAttr  ::= 'resource="' URI-reference '"'
 [10] Qname         ::= [ NSprefix ':' ] name
 [11] URI-reference ::= string, interpreted per [URI]
 [12] IDsymbol      ::= (any legal XML name symbol)
 [13] name          ::= (any legal XML name symbol)
 [14] NSprefix      ::= (any legal XML namespace prefix)
 [15] string        ::= (any XML text, with "<", ">", and "&" escaped)
```

RDF 元素是一个简单的包装层,它标记了一个 XML 文档中的边界,在边界之间的内容可被明确地映射成 RDF 数据模型实例。如果从应用的上下文中可以知道内容是 RDF,则 RDF 元素是可选的。

上例中的句子 Ora Lassila is the creator of the resource http://www.w3.org/Home/Lassila,可表示成如下的 RDF/XML:

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator>Ora Lassila</s:Creator>
  </rdf:Description>
</rdf:RDF>
```

这里名字空间前缀"s"指的是一个特定的名字空间前缀。它是由该 RDF 表示式的作者选择的,其 XML 名字空间说明的定义如下:

```
xmlns:s=http://description.org/schema/
```

包含名字空间描述的完整 XML 文档如下:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator>Ora Lassila</s:Creator>
  </rdf:Description>
</rdf:RDF>
```

基本的缩写语法

虽然序列化语法很清楚地表示了 RDF 的结构,通常人们又希望使用一种更紧凑的 XML 形式。RDF 缩写语法实现了这一点。其另外的优点是它允许文档遵从某一良好建构的 XML DTD 从而可直接解释为 RDF 模型。

对基本的序列化定义了三种缩写形式。第一种用于在 Description 中属性没有重复而且这些属性的值是文字。在这种情况下,属性可表示为 Description 元素的一个 XML Attribute。前一个例子如下:

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila"
                   s:Creator="Ora Lassila" />
</rdf:RDF>
```

这种缩写的另一个例子是：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org">
    <s:Publisher>World Wide Web Consortium</s:Publisher>
    <s:Title>W3C Home Page</s:Title>
    <s:Date>1998-10-03T02:27</s:Date>
  </rdf:Description>
</rdf:RDF>
```

等同于：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org"
       s:Publisher="World Wide Web Consortium"
       s:Title="W3C Home Page"
       s:Date="1998-10-03T02:27"/>
</rdf:RDF>
```

第二种 RDF 缩写形式在嵌套的 Description 元素上做工作。它用于特定的声明——声明的客体是另一个资源而且该资源所有属性的值都是字符串。在这种情况下，使用了一个类似于 XML 元素到 XML 属性的转换：嵌套 Description 中资源的属性可写成 Description 包含在内的 propertyElt 元素的 XML 属性。如句子：

The individual referred to by employee id 85740 is named Ora Lassila and has the email address lassila@w3.org. The resource http://www.w3.org/Home/Lassila was created by this individual.

用明确的序列化形式写成 RDF/XML 就是：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator rdf:resource="http://www.w3.org/staffId/85740"/>
  </rdf:Description>

  <rdf:Description about="http://www.w3.org/staffId/85740">
    <v:Name>Ora Lassila</v:Name>
    <v:Email>lassila@w3.org</v:Email>
  </rdf:Description>
</rdf:RDF>
```

明确地表示第二个资源为第一个资源所用，相同的表示式可写成：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
```

```
    <s:Creator>
      <rdf:Description about="http://www.w3.org/staffId/85740">
      <v:Name>Ora Lassila</v:Name>
      <v:Email>lassila@w3.org</v:Email>
      </rdf:Description>
    </s:Creator>
  </rdf:Description>
</rdf:RDF>
```

使用第二种基本缩写语法，内部 Description 元素和其包含的属性表示式可被写成 Creator 元素的属性：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator rdf:resource="http://www.w3.org/staffId/85740"
       v:Name="Ora Lassila"
       v:Email="lassila@w3.org" />
  </rdf:Description>
</rdf:RDF>
```

第三种基本的缩写用于 Description 元素包含一个 type 属性的情况，这时定义在模式中对应于 type 属性的值的资源类型可直接用做一个元素名。比如使用前一个 RDF 片段，如果想增加一个事实，资源 http://www.w3.org/staffId/85740 表示一个人的实例，用完整的序列化语法写出来就是：

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator>
      <rdf:Description about="http://www.w3.org/staffId/85740">
      <rdf:type resource="http://description.org/schema/Person"/>
      <v:Name>Ora Lassila</v:Name>
      <v:Email>lassila@w3.org</v:Email>
      </rdf:Description>
    </s:Creator>
  </rdf:Description>
</rdf:RDF>
```

使用缩写就是：

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator>
      <s:Person about="http://www.w3.org/staffId/85740">
      <v:Name>Ora Lassila</v:Name>
      <v:Email>lassila@w3.org</v:Email>
```

```
      </s:Person>
    </s:Creator>
  </rdf:Description>
</rdf:RDF>
```

基本缩写语法替代基本序列化语法的[2]和[6]用 EBNF 表示就是：

```
[2a] description    ::= '<rdf:Description' idAboutAttr? propAttr* '/>'
                      | '<rdf:Description' idAboutAttr? propAttr* '>'
                            propertyElt* '</rdf:Description>'
                      | typedNode
[6a] propertyElt    ::= '<' propName '>' value '</' propName '>'
                      | '<' propName resourceAttr? propAttr* '/>'
[16] propAttr       ::= propName '="' string '"'
                        (with embedded quotes escaped)
[17] typedNode      ::= '<' typeName idAboutAttr? propAttr* '/>'
                      | '<' typeName idAboutAttr? propAttr* '>'
                          property* '</' typeName '>'
```

模式和名字空间

当用自然语言描述一条语句，我们使用表达特定含意的词。在 RDF 应用中，含意对理解声明和建立正确的处理是至关重要的。声明的作者和读者应对所用的术语有相同的理解。在全球范围的媒介，如 WWW 上依赖一个对概念的共同文化上的理解明显不够。

RDF 中的含意通过对模式的引用来表达。可以认为模式是一种字典，一个模式定义将用于 RDF 声明中的术语并给予其特定的含意。RDF 可以使用多种模式形式，包括定义在单独文档[RDFSchema]中的特定形式，其具有一些特别的特性帮助使用 RDF 完成自动的任务。

为了避免对同一术语理解上的混乱而且可能是冲突，RDF 使用了 XML 名字空间机制。

2.2.2 容器

在构建基本 RDF 模型时，经常会涉及到一组资源，比如一项工作由多个人建立，列出某课程的学生或一个包中的所有软件模块，可以采用 RDF 容器容纳这些资源或文字的列表。

2.2.2.1 容器模型

RDF 定义了三种类型的容器对象：

包(Bag)：资源或文字的无序列表。包用来说明具有多个值的属性且值的排列顺序没有意义。允许有重复的值。

序列(Sequence)：资源和文字的有序列表。序列用于说明一个属性具有多个值且值的顺序很重要，比如序列可以是值的一个字母顺序。允许有重复的值。

替代值(Alternative)：表示可替代属性的(单个)值的资源或文字的列表。替代值可用于提供工作名称的可替代的语言译文，或者提供资源在 Internet 上的镜像站点。一个属性其值为替代值集合就表明它能选择列表中的任一项作为其值。

容器一个通常的用法是作为一属性的值。当以这种方法使用时，声明仍具有单个声明对象，不管容器中的成员有多少，容器资源自身就是声明的对象。

比如要表示句子：

The students in course 6.001 are Amy, Tim, John, Mary, and Sue.

其 RDF 模型就是图 2.2。

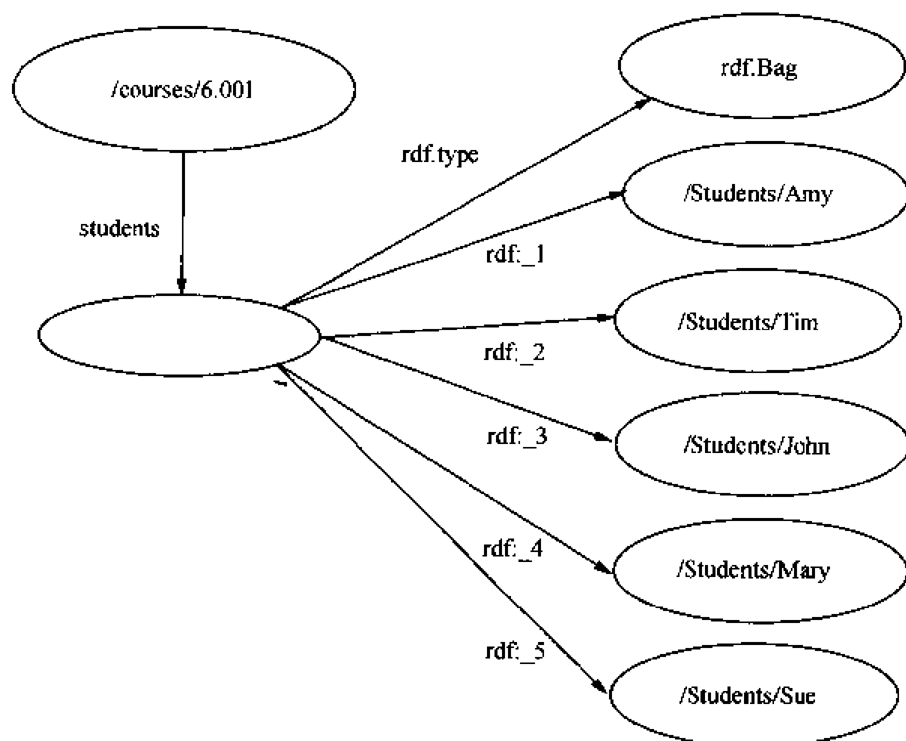


图 2.2 简单包容器

句子 The source code for X11 may be found at ftp.10.org, ftp.cs.purdue.edu, or ftp.eu.net.的 RDF 模型是图 2.3。

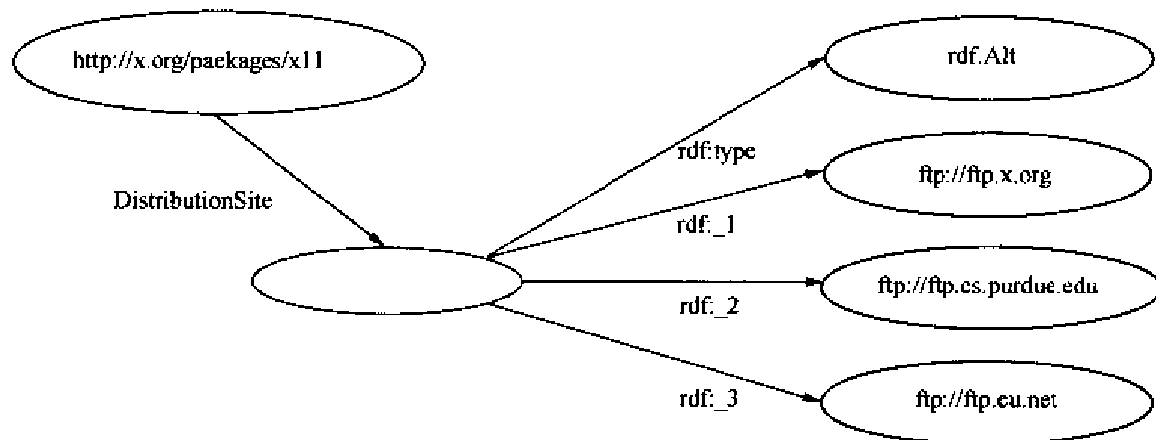


图 2.3 简单的替代器容器

2.2.2.2 容器语法

RDF 容器语法采用如下的形式：

[18] container ::= sequence | bag | alternative

```
[19] sequence        ::= '<rdf:Seq' idAttr? '>' member * '</rdf:Seq>'
[20] bag             ::= '<rdf:Bag' idAttr? '>' member * '</rdf:Bag>'
[21] alternative     ::= '<rdf:Alt' idAttr? '>' member + '</rdf:Alt>'
[22] member          ::= referencedItem | inlineItem
[23] referencedItem  ::= '<rdf:li' resourceAttr '/>'
[24] inlineItem      ::= '<rdf:li>' value '</rdf:li>'
```

容器可用于允许使用 Description 的任何地方。

```
[1a] RDF             ::= '<rdf:RDF>' obj * '</rdf:RDF>'
[8a] value           ::= obj | string
[25] obj             ::= description | container
```

一个 Alt 容器至少需要具有一个成员。该成员将由属性_1 识别,是默认值或偏好值。

例子

句子 The students in course 6.001 are Amy, Tim, John, Mary, and Sue. 写成 RDF/XML 就是:

```
<rdf:RDF>
  <rdf:Description about="http://mycollege.edu/courses/6.001">
    <s:students>
      <rdf:Bag>
      <rdf:li resource="http://mycollege.edu/students/Amy"/>
      <rdf:li resource="http://mycollege.edu/students/Tim"/>
      <rdf:li resource="http://mycollege.edu/students/John"/>
      <rdf:li resource="http://mycollege.edu/students/Mary"/>
      <rdf:li resource="http://mycollege.edu/students/Sue"/>
      </rdf:Bag>
    </s:students>
  </rdf:Description>
</rdf:RDF>
```

句子 The source code for X11 may be found at ftp.10.org, ftp.cs.purdue.edu, or ftp.eu.net. 写成 RDF/XML 就是:

```
<rdf:RDF>
  <rdf:Description about="http://10.org/packages/X11">
    <s:DistributionSite>
      <rdf:Alt>
      <rdf:li resource="ftp://ftp.10.org"/>
      <rdf:li resource="ftp://ftp.cs.purdue.edu"/>
      <rdf:li resource="ftp://ftp.eu.net"/>
      </rdf:Alt>
    </s:DistributionSite>
  </rdf:Description>
</rdf:RDF>
```

2.2.2.3 分布的指示:关于容器成员的声明

容器结构会产生一个关于声明的问题:当一个声明引用一个集合时,声明描述的"事物"是什么呢?换句话说,声明所涉及的客体是什么呢?声明是描述容器本身还是描述容器的成员?描述的客体(XML 语法中由 about 属性指出)在 RDF 中称为指示(Referent)。

RDF 中的指示用 aboutEach 来表示,分布的指示允许在 RDF 描述中共享结构。语法如下:

```
[3a] idAboutAttr    ::= idAttr | aboutAttr | aboutEachAttr
[26] aboutEachAttr  ::= 'aboutEach="' URI-reference '"'
```

如有这样的包定义

```
<rdf:Bag ID="pages">
  <rdf:li resource="http://foo.org/foo.html" />
  <rdf:li resource="http://bar.org/bar.html" />
</rdf:Bag>
```

则

```
<rdf:Description aboutEach="#pages">
  <s:Creator>Ora Lassila</s:Creator>
</rdf:Description>
```

等同于:

```
<rdf:Description about="http://foo.org/foo.html">
  <s:Creator>Ora Lassila</s:Creator>
</rdf:Description>
<rdf:Description about="http://bar.org/bar.html">
  <s:Creator>Ora Lassila</s:Creator>
</rdf:Description>
```

2.2.2.4 由 URI 方式定义的容器

元数据一个经常的用法就是对"本站点上的所有页"或者"本站点这一分支的所有页"做出声明。在许多情况下,将这些资源明确列出并将其确定为一个容器的成员是不切实际且不必要的。因此 RDF 定义了第二个分布指示,它是一种速记语法,表示一个包"Bag"的一个实例,其成员被定义为以一特殊的字符串开始的资源标志符指示的所有资源:

```
[26a] aboutEachAttr  ::= 'aboutEach="' URI-reference '"'
                       | 'aboutEachPrefix="' string '"'
```

2.2.2.5 容器与重复属性

一个资源可能具有多个声明,其谓词相同(即使用相同的属性)。这和其客体是一个包含了多个成员的容器的单个声明是不同的。在特定的条件下选择使用哪一个声明由模式设计人员决定,如何声明由特定的 RDF 声明编写人员决定。

2.2.3 关于声明的声明

除了作出关于 Web 资源的声明，RDF 也能用于做出关于其他 RDF 声明的声明，称为更高状态的声明。为了作为关于另一个声明的声明，实际上必须建立一个原始声明的模型，这一模型是一个新的资源，可对其附加另外的属性。

一个声明的模型是我们对建模后的声明做出“更高状态的声明”时所需的资源。

比如，句子“Ora Lassila is the creator of the resource http://www.w3.org/Home/Lassila.”RDF 将其看做一个事实。而句子“Ralph Swick says that Ora Lassila is the creator of the resource http://www.w3.org/Home/Lassila.”没有关于资源 http://www.w3.org/Home/Lassila 的任何信息，实际上只是表达了关于 Ralph 做出一个声明这个事实。为了用 RDF 表示这个事实，必须将最初的声明建模为一个具有四个属性的资源。这一过程被知识表示团体正式地称为具体化(Reification)。一个声明的模型被称为具体化的声明(Reified Statement)。

为对声明建模，RDF 定义了下述属性：

主体：主体属性确定由建模了的声明所描述的资源，即主体属性的值是对其做出最初声明的资源(这个例子中就是 http://www.w3.org/Home/Lassila)。

谓词：谓词属性确定建模后声明中的初始属性。谓词属性的值是最初声明中表示特定属性的资源(本例中即 Creator)。

客体：客体属性确定建模后声明中的属性值。其值是初始声明中对象(本例中即“Ora Lassila”)。

类型：类型属性的值描述了新资源的类型。所有具体化的声明都是 RDF:Statement 的实例，即它们有一个类型值其客体是 RDF:Statement。类型属性可更一般地用于说明任何资源的类型。

属性 bagID 指定容器资源的资源 ID：

```
[26] description   ::= '<rdf:Description' idAboutAttr? bagIDAttr? propAttr* '/>'
                     | '<rdf:Description' idAboutAttr? bagIDAttr? propAttr* '>'
                           propertyElt* '</rdf:Description>'
[27] bagIDAttr     ::= 'bagID="' IDsymbol '"'
```

2.2.4 RDF 正式模型

数据模型有三种表示法：作为三元组，作为图和用 XML 表示。这些表示都具有相同的含义，表示间的映射不会以任何方式约束实现中使用的内部表示。

RDF 数据模型正式定义如下：

(1) 有一个集合称为资源(Resources)。

(2) 有一个集合称为文字(Literals)。

(3) Resources 有一个子集称为属性(Properties)。

(4) 有一个集合称为声明(Statements)，其每个元素是一个三元组，形式为

{pred, sub, obj}

其中 pred 是一个属性(Properties 的成员)，sub 是一个资源(Resources 的成员)，obj 是一个资源或文字(Literals 的成员)。

(5) Properties 有一个元素是 RDF:type。

(6) 形式为{RDF:type, sub, obj}声明的成员必须满足下列条件:sub 和 obj 是 Resources 的成员。[RDFSchema]对类型的使用增加额外的限制。

(7) Resources 有一个不包含在 Properties 中的元素 RDF:Statement。

(8) Properties 有三个元素即 RDF:predicate,RDF:subject 和 RDF:object。

(9) 声明的三元组{pred, sub, obj}的具体化是 Resources 的一个元素 r,表示具体化的三元组及声明的元素为 s1,s2,s3 和 s4。

s1: {RDF:predicate, r, pred}
s2: {RDF:subject, r, subj}
s3: {RDF:object, r, obj}
s4: {RDF:type, r, [RDF:Statement]}

(10) Resources 有三个不包含在 Properties 中的元素,即 RDF:Seq,RDF:Bag 和 RDF:Alt。

(11) 对应于序数 Properties 的子集称为 Ord。涉及 Ord 的元素时使用 RDF:_1,RDF:_2,RDF:_3……

2.2.5 RDF 正式语法

下面完整地描述了前面所述的 RDF 的巴科斯范式——RDF 的正式语法。

```
[1] RDF            ::= ['<rdf:RDF>'] obj* ['</rdf:RDF>']
[2] obj            ::= description | container
[3] description    ::= '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '/>'
                      | '<rdf:Description' idAboutAttr? bagIdAttr? propAttr* '>'
                            propertyElt* '</rdf:Description>'
                      | typedNode
[4] container      ::= sequence | bag | alternative
[5] idAboutAttr    ::= idAttr | aboutAttr | aboutEachAttr
[6] idAttr         ::= ' ID="' IDsymbol '"'
[7] aboutAttr      ::= ' about="' URI-reference '"'
[8] aboutEachAttr  ::= ' aboutEach="' URI-reference '"'
                      | ' aboutEachPrefix="' string '"'
[9] bagIdAttr      ::= ' bagID="' IDsymbol '"'
[10] propAttr      ::= typeAttr
                      | propName '="' string '"' (with embedded quotes escaped)
[11] typeAttr      ::= ' type="' URI-reference '"'
[12] propertyElt   ::= '<' propName idAttr? '>' value '</' propName '>'
                      | '<' propName idAttr? parseLiteral '>'
                            literal '</' propName '>'
                      | '<' propName idAttr? parseResource '>'
                            propertyElt* '</' propName '>'
                      | '<' propName idRefAttr? bagIdAttr? propAttr* '/>'
[13] typedNode     ::= '<' typeName idAboutAttr? bagIdAttr? propAttr* '/>'
                      | '<' typeName idAboutAttr? bagIdAttr? propAttr* '>'
```

```
                                   propertyElt * '</' typeName '>'
[14] propName          ::= Qname
[15] typeName          ::= Qname
[16] idRefAttr         ::= idAttr | resourceAttr
[17] value             ::= obj | string
[18] resourceAttr      ::= ' resource="' URI-reference '"'
[19] Qname             ::= [ NSprefix ':' ] name
[20] URI-reference     ::= string, interpreted per [URI]
[21] IDsymbol          ::= (any legal XML name symbol)
[22] name              ::= (any legal XML name symbol)
[23] NSprefix          ::= (any legal XML namespace prefix)
[24] string            ::= (any XML text, with "<", ">", and "&" escaped)
[25] sequence          ::= '<rdf:Seq' idAttr? '>' member * '</rdf:Seq>'
                         | '<rdf:Seq' idAttr? memberAttr * '/>'
[26] bag               ::= '<rdf:Bag' idAttr? '>' member * '</rdf:Bag>'
                         | '<rdf:Bag' idAttr? memberAttr * '/>'
[27] alternative       ::= '<rdf:Alt' idAttr? '>' member - '</rdf:Alt>'
                         | '<rdf:Alt' idAttr? memberAttr? '/>'
[28] member            ::= referencedItem | inlineItem
[29] referencedItem    ::= '<rdf:li' resourceAttr '/>'
[30] inlineItem        ::= '<rdf:li' '>' value </rdf:li>'
                         | '<rdf:li' parseLiteral '>' literal </rdf:li>'
                         | '<rdf:li' parseResource '>' propertyElt * </rdf:li>'
[31] memberAttr        ::= ' rdf:_n="' string '"' (where n is an integer)
[32] parseLiteral      ::= ' parseType="Literal"'
[33] parseResource     ::= ' parseType="Resource"'
[34] literal           ::= (any well-formed XML)
```

2.3 RDF 模式(Schema)

综上所述,RDF 数据模型,就命名属性和值而言,为描述资源间相互关系定义了一种简单的模型。可以认为 RDF 属性是资源的属性,对应于传统的属性与其值的组合,RDF 属性同样代表了资源间的关系。因此,RDF 数据模型和实体-关系图类似。但是,RDF 数据模型没有提供机制来说明这些属性,也没有提供任何机制来定义这些属性和其他资源间的关系。RDF 模式用于完成这些任务。

资源描述团体需要一种能力来表示某类资源的某些事情。比如:为描述传记资源,通常有描述性属性包括"作者"、"标题"和"主题"。对于数字确认,通常需要属性如"核对总数"和"授权"等属性。这些属性和其相应的语义说明通常在 RDF 背景下作为 RDF 模式定义。模式不仅定义资源的属性(如标题、作者、主题、大小、颜色等),也可定义被描述的某一类资源(书、Web 页面、人、公司等)。

RDF 模式规范并不说明描述性元素的词汇,如"作者",而是说明定义这些元素所需的机制,定义可能与之一起被使用的资源的类,约束类和关系的可能联合,检测是否违背了这些约

束。简单地说,RDF 模式机制提供了一个应用于 RDF 模型中的基本类型系统。它定义了资源和属性如 rdfs: Class 和 rdfs: subClassOf 及其被使用于说明特定应用的模式。

2.3.1 范围

RDF 模式规范不以理论问题为目标,而是解决少量直接的问题。它的创建者希望具有相似特征的其他问题(其中一些在下面的例子中说明)同样也可以使用规范中描述的基本类。

RDF 模式规范直接出于对下列问题的考虑。

2.3.1.1 Internet 内容选择的平台(PICS)

RDF 模型和语法足以表示 PICS 标记,但是它不提供用于一般目的的从 PICS 评价系统到 RDF 表示的映射。

2.3.1.2 简单的 Web 元数据

RDF 一个典型的应用是对 Web-页面的描述,这是 Dublin Core[DC]元数据创始的基本目标之一。Dublin Core 元素集合是 15 个元素的集合,这些元素被认为广泛应用于描述 Web 资源,使其能被发现。Dublin Core 对 RDF 开发影响最大。Dublin Core 开发中一个重要的考虑就是不仅允许简单的描述,同时提供限制修饰描述的能力,从而提供特定领域的细节和精确描述。

RDF 模式规范提供了一种机器能理解的系统来为描述性词汇,如 Dublin Core 定义模式。它允许设计者说明资源类型的类和属性来传达这些类的描述,这些属性和类之间的关系以及对类、属性和值可能存在的结合的约束。

2.3.1.3 站点映像和概念导航

站点映像是一个 Web 站点的层次描述。主题分类法是一种分类系统,可以被内容创建者或可信任的第三方用于组织或分类 Web 资源。RDF 模式规范提供了一种机制定义这些应用所需的词汇。

就指定的概念间的关系而言,辞典和图书馆的分类模式是代表主题分类的层次系统的最常见的例子。RDF 模式规范提供了足够的资源来创建表示辞典(其他图书馆分类系统)逻辑结构的 RDF 模型。

2.3.1.4 P3P

针对私有偏好工程(P3P)的 W3C 平台除了规范其交换结构数据的语法外,还说明了构建声明的语法。这些声明是关于站点数据收集的实践和从这些实践训练得出的个人偏好。

虽然个人数据收集实践已经对 P3P 中使用了特定应用的 XML 标记集进行了描述,RDF 模式为该数据使用一个通用的元数据模型也是有好处的。P3P 策略的结构可被解释为一个 RDF 模型。使用一个元数据模式来描述私有实践描述的语义,这将允许资源发现过程中在查询里使用私有实践数据和其他元数据,也将允许一个通用的软件主体使用与其他描述元数据的相同技术来作用于私有元数据上。P3P 的扩展描述中有关于站点收集的特定数据元素,使用 RDF 模式能进一步说明如何使用这些数据元素。

2.3.2 类和属性

RDF 模式由 RDF 模型和语法规范[RDFMS]中所描述的数据模型来表达。模式描述语言只不过是资源和属性的集合，这些属性是由 RDF 模式规范和使用 RDF 模式机制的每个 RDF 模型的隐含部分所定义的。

RDF 模式规范中指定 RDF 模式机制是一个 RDF 资源(包括类和属性)的集合和对它们关系的约束。抽象的 RDF 模式核心词汇能被用于做出 RDF 声明，定义和描述特定应用的词汇如 Dublin Core 元素集。

2.3.2.1 类型系统

RDF 模式是能被用于描述其他 RDF 资源(包括属性)的 RDF 资源的集合，它们定义了特定应用的 RDF 词汇。这里核心模式词汇在一个称为“rdfs”的名字空间中定义，并以 URI 引用 http://www.w3.org/2000/01/rdf-schema# 来标识。这里同样使用前缀“rdf”来指示核心 RDF 名字空间 http://www.w3.org/1999/02/22-rdf-syntax-ns#。

如 RDF 模型和语法规范中所描述的，资源可能是一个或多个类的实例，这由 rdf:type 属性来指示。类自身通常以层次方式组织，比如说类 Dog 可被认为是类 Mammal 的子类，而类 Mammal 是类 Animal 的子类，这意味着任何 rdf:type Dog 的资源同样可认为是 rdf:type Animal 的子类。RDF 模式规范定义了一个属性 rdf:subClassOf 来解释这些类间的关系。

图 2.4 说明了类、子类和资源的概念。类用一个圆角的方框表示，资源用一个大点表示。在如下的图中，箭头从一个资源指向所定义的类。一个子类表现为一个圆角方框(子类)完全包含在另一个圆角方框(超类)中。如果一个资源在一个类中，则存在该资源一个明确或隐含的 rdf:type 属性，其值是定义包含类的资源。这些属性在图 2.5 中表现为弧。

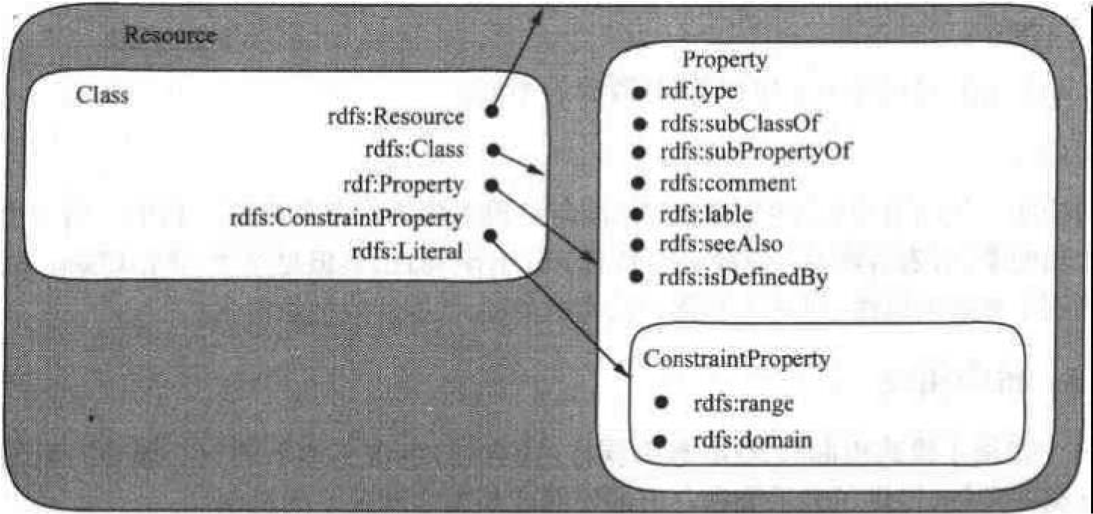


图 2.4 作为集合和元素的类和资源

图 2.5 表示和图 2.4 所示相同的类层次信息，但它是使用 RDF 数据模型的“节点和弧”来实现的。如果一个类是另一个类的子类，那么存在一条 rdfs: subClass 弧从表示第一个类的节点指向表示第二个类的节点。如果一个资源是另一个资源的实例，那么存在一条 rdf: type 弧从资源指向表示类的节点。这里没有画出所有的弧，而只画出指向关系最密切的类的弧，依赖于 rdfs: subClassOf 关系的传递来提供其余的弧。

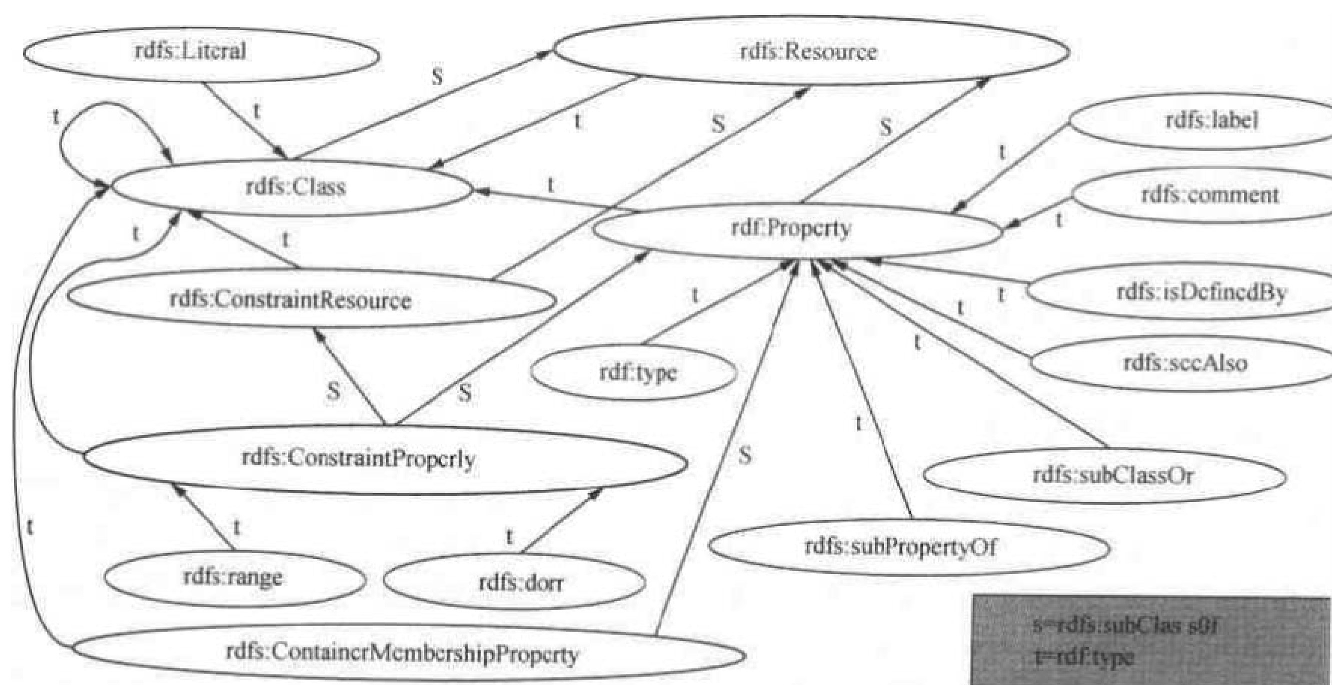


图 2.5　针对 RDF 模式的类层次

2.3.2.2　核心类

下述资源是作为 RDF 模式词汇一部分的核心类。每一个运用 RDF 模式名字空间的 RDF 模型都(隐含地)包含它们。

rdfs: Resource

由 RDF 表达式所描述的所有东西都被称为资源,并被认为是 rdfs: Resource 类的实例。

rdf: Property

rdf: Property 表示称为属性的 RDF 资源的子集。

rdfs: Class

它对应于一般的类型或分类的概念,与面向对象编程语言中类的概念相似。当一个模式定义了一个新的类时,表示该类的资源必须具有 rdf: type 属性,其值是资源 rdfs: Class。RDF 类可被定义为表示所有的东西,如 Web 页面、人、文档类型、数据库或抽象概念。

2.3.2.3　核心属性

每一个使用了模式机制的 RDF 模型同样(隐含地)包含了如下的核心属性。它们是 rdf: Property 类的实例,提供了表示类和它们的实例或超类之间关系的机制。

rdf: type

它指示资源是一个类的成员,因此具有类的成员所希望具有的所有特征。当一个资源具有 rdf:type 属性,并且其值是某些特定的类时,就认为资源是指定类的一个实例。某些资源的 Rdf:type 属性的值必须是 rdfs:Calss 实例的另一资源。人们所知的 rdfs:Class 资源其本身是 rdf:type rdfs:Class 的资源。单个的类(如"Dog")通常具有属性 rdf:type,其值是 rdfs:Class(或 rdfs:Class 的某一子类)。一个资源可以是多个类的实例。

rdfs: subClassOf

这一属性说明类间的一个子类/超类关系。Rdfs:subClassOf 属性是可传递的。如果类 A 是某一更大的类 B 的子类,B 是 C 的子类,则 A 同样是 C 的子类。因此,作为类 A 实例的资源将同样是 C 的实例,因为 A 同时是 B 和 C 的子集。只有 rdfs:Class 的实例能具有 rdfs:subClassOf 属性,且属性值总是 rdf:type rdfs:Class。一个类可以是多个类的子类。

一个类不能说明为自己的子类,也不能是它自己子类的子类。

下面是一个非常简单的例子(图 2.6),表示了如下的类层次。首先定义一个类 MotorVechicle。然后再定义 MotorVechicle 的三个子类,即 PassengerVechicle、Truck 和 Van。然后定义一个类 Minivan,它同时是 Van 和 PassengerVechicle 的子类。

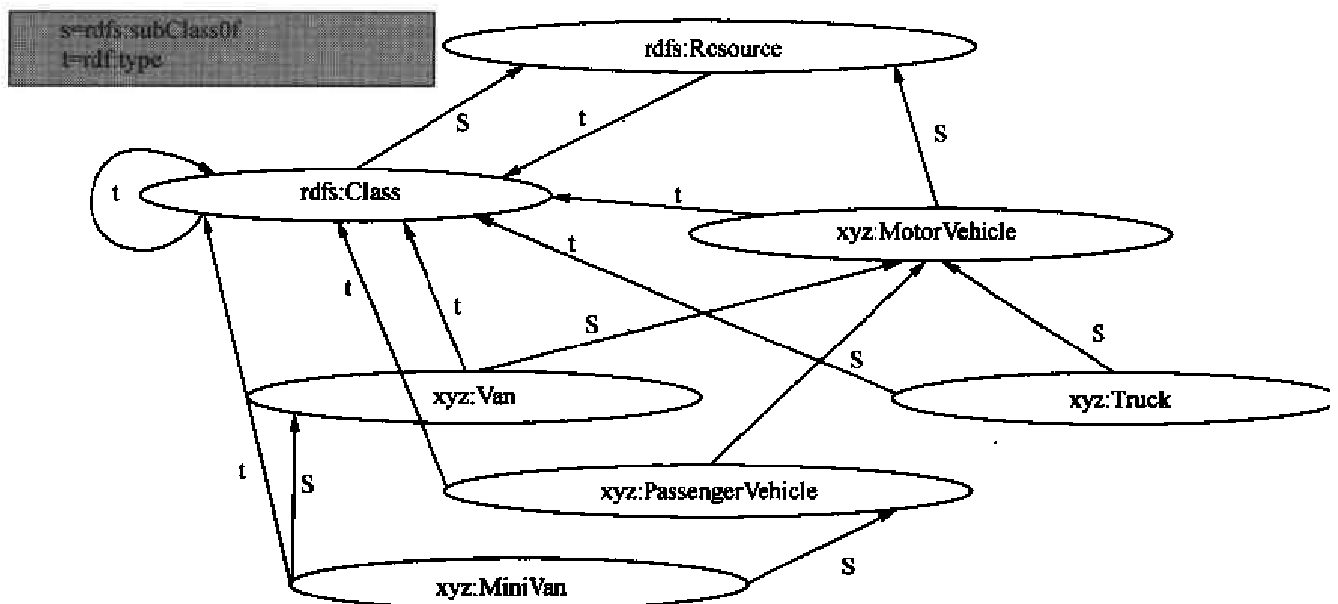


图 2.6 rdfs:subClassOf 的例子

这里的 RDF/XML 使用了基本的 RDF 语法。

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<! -Note: this RDF schema would typically be used in RDF instance data
    by referencing it with an XML namespace declaration, for example
    xmlns:xyz="http://www.w3.org/2000/03/example/vehicles#".   This allows
    us to use abbreviations such as xyz:MotorVehicle to refer
    unambiguously to the RDF class 'MotorVehicle'. -->

 <rdf:Description ID="MotorVehicle">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
 </rdf:Description>
```

```
<rdf:Description ID="PassengerVehicle">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="Truck">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="Van">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="MiniVan">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#Van"/>
      <rdfs:subClassOf rdf:resource="#PassengerVehicle"/>
  </rdf:Description>
</rdf:RDF>
```

rdfs: subPropertyOf

属性 rdfs: subPropertyOf 是 rdfs: Property 的一个实例,它用于说明一个属性是另一个属性的特殊化。一个属性可能是零个、一个或多个属性的特殊化。如果某一属性 P2 是另一个更一般化属性 P1 的 subPropertyOf,并且如果一个资源 A 具有 P2 属性,其值是 B,这就隐含着资源 A 同样具有属性 P1,其值是 B。

一个属性不能说明为自己的子属性,也不能是它自己子属性的子属性。

下面是一个例子。如果属性 biologicalFather 是另一个更广泛的属性 biologicalParent 的子属性,并且如果 Fred 是 John 的 biologicalFather,这就隐含着 Fred 同样是 John 的 biologicalParent。

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdf:Description ID="biologicalParent">
 <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
</rdf:Description>
<rdf:Description ID="biologicalFather">
 <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:subPropertyOf rdf:resource="#biologicalParent"/>
</rdf:Description>
</rdf:RDF>
```

rdfs: seeAlso

属性 rdfs: seeAlso 说明一个资源可能提供关于主题资源的附加信息。这一属性可以使用 rdfs: subPropertyOf 来说明,从而更准确地指示客体资源具有的关于主体资源的信息的状态。客体和主体资源强制为只能是类 rdfs: Resource 的实例。

rdfs: isDefinedBy

属性 rdfs: isDefinedBy 是 rdfs: seeAlso 的子属性,指示了定义主体资源的资源。和使用 rdf: seeAlso 一样,这一属性能被应用于 rdfs: Resource 的任何实例,并且可具有任何 rdfs: Resource 值。

人们普遍期望的用法是给出该模式所定义的一个属性或类的名字,识别一个 RDF 模式。虽然典型的 XML 名字空间说明提供 URI,在那里定义了词汇资源,但在某些情况下仍需要另外的信息。

比如说,约束如:

<rdfs:subPropertyOf rdf:resource="http://purl.org/dc/elements/1.0/Creator"/>

没有指示包括词汇术语 Creator 的模式的 URI(即 http://purl.org/dc/elements/1.0/)。

在这些情况下,属性 rdfs: isDefinedBy 能被用于明确的表示该信息。当名字空间的 URI 和其组件没有明显的关系时,如在使用模式 GUID 或 MD-5 组合信息来识别它们时,这一方法同样有效。

2.3.3 约束

RDF 模式规范引入了一个 RDF 词汇来声明使用 RDF 数据中属性和类的约束。比如一个 RDF 模式可能描述对属性值类型的限制,让其对某一属性有效;或者描述对类的属性的限制,对这些类这些属性是有意义的。

RDF 模式提供了一种机制描述这些约束,但是没有说明一个应用是否或者必须如何处理约束信息。例如,RDF 模式能断言一个 author 属性被用于指示为类 Person 的成员的资源。同时,它没有说明一个应用在处理该类信息是否或者如何做。我们希望不同的应用以不同的方式使用这些约束,如一个合法检查程序查找错误,一个交互编辑器建议可能的合法值。

RDF 模式能表示连接来自多个独立开发模式的词汇术语的约束。因为 URI 引用用于识别类和属性,故有可能创建新的属性,其 domain 或 range 约束引用定义在另一名字空间的约束。

下面的约束在 RDF 模式 1.0 中说明 rdfs: domain 和 rdfs:range 如何对属性用法进行约束,规则为属性 rdfs: subPropertyOf 和 rdfs: subClassOf 不应当形成循环,同时任何使用 rdfs: ConstraintResource 扩展机制进行定义的进一步约束同样适用。不同的应用在处理 RDF 约束时可能表现出不同的行为。

约束的一些例子包括:

一个属性的值应当是一个指定类的资源,这被称为是一个范围(Range)约束。比如一个应用于 author 属性的范围约束可能表示了 author 属性的值必须是类 Person 的一个资源。

一个属性可能用在某一类资源上,这被称为领域(Domain)约束。比如 author 属性只能作为类 Book 的一个实例的一个资源而被创建。

RDF 规范没有列举应用于 RDF 词汇描述的所有可能的约束形式，而只定义了一些基本的约束机制，同时一个扩展功能可允许随后新型约束的增加。

虽然 RDF 数据模型允许不明确的属性(如 rdf:type 属性)作为文字出现(原子值)，我们仍然认为这些实体是类的成员(如字符串"John Smith"被认为是类 rdfs: Literal 的一个成员)。

2.3.3.1 核心约束

rdfs:ConsrtaintResource

这一资源定义了 rdfs: Resource 的一个子类，其实例是涉及到表达约束的 RDF schema 模型。该类的目的是提供一种机制让 RDF 处理器评估其使用与 RDF 模型有关的约束信息的能力。因为模式规范没有提供一种动态发现约束新形式的机制，RDF 模式 1.0 处理器遇到先前不知名的 rdfs:ConstrainResource 的实例时就确认它没有资格决定这些约束的含意。

rdfs:ConstraintProperty

该资源定义了 rdf:Property 的一个子类，其所有的实例是用于说明约束的属性。该类是 rdfs:ConstraintResource 的子类，对应于代表属性的类的子集。rdfs:domain 和 rdfs:rannge 都是 rdfs: ConstraintProperty 的实例。

rdfs: range

ConstraintProperty 的一个实例，用于指示类，一个属性的值必须是这些类的成员。属性 range 的值通常是一个类。范围约束只应用于属性。

一个属性最多具有一个范围属性。没有范围也是可能的，这种情况下属性值的类是无约束的。

Rdfs: range 的 rdfs: domain 是类 rdf: Property。这表明应用于资源的 range 属性本身是属性。

Rdfs: range 的 rdfs: range 是类 rdf: Class。这表明作为范围属性的值的任何资源将是一个类。

rdfs: domain

ConstraintProperty 的一个实例，用于指示一个类，一个属性可应用于其成员上。

一个属性可具有零个、一个或多个类作为其 domain。如果没有 domain 属性，它可和任何资源一起使用。如果恰好有一个 domain 属性，它只可用于该类的实例(该类是 domain 属性的值)。如果有多个 domain 属性，约束属性能和任何这些类的实例一起使用(这些类是这些 domain 属性的值)。

Rdfs: doamin 的 rdfs: domain 是类 rdf: Property。这表明 domain 属性应用于属性资源上。

Rdfs: range 的 rdfs: domain 是类 rdf: Class。这表明作为领域属性的值的任何资源将是一个类。

RDF 模式使用约束属性来约束其自身的属性如何使用。这些属性在下面的图 2.7 中表示。具有粗体轮廓的结点是 rdfs: Class 的实例。

2.3.3.2 例子

继续前一个 MotorVehicle 的例子，在这个例子中，定义了两个属性：RegisterdTo 和

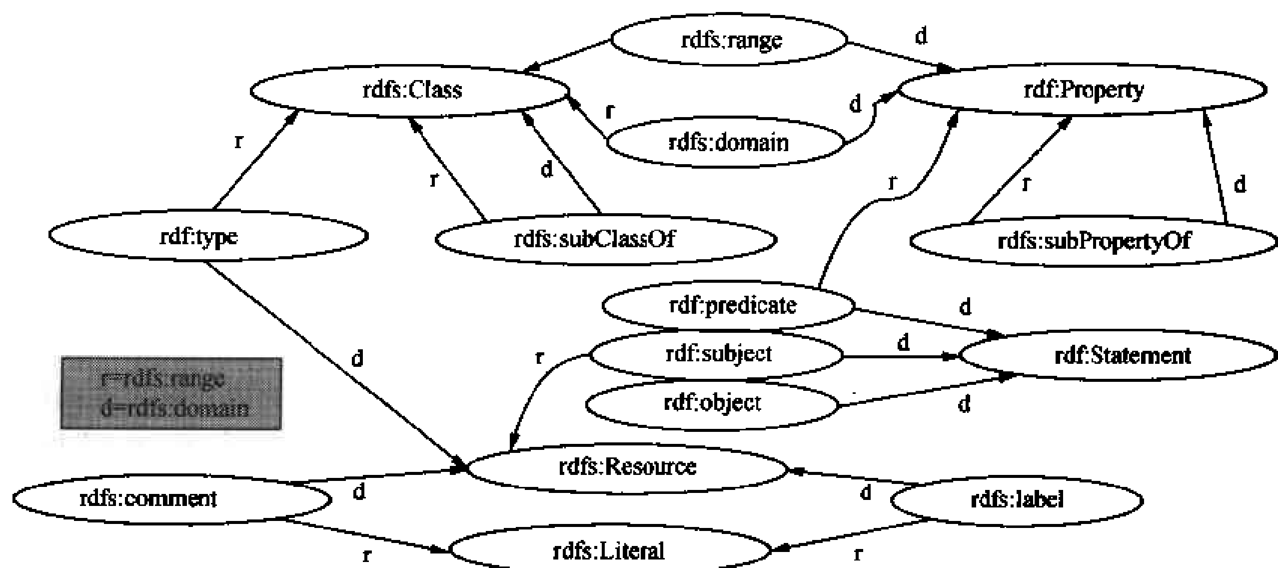


图 2.7 RDF 模式中的约束

rearSeatLeaRoom。RegisterdTo 属性可应用于任何 MotorVehicle,其值是一个人。在这个例子中,rearSeatLeaRoom 只应用于 Minivans 和 PassengerVehicles,其值是一个数字。

```
<rdf:RDF xml:lang="en"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdf:Description ID="registeredTo">
 <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:domain rdf:resource="#MotorVehicle"/>
 <rdfs:range rdf:resource="#Person"/>
 </rdf:Description>
<rdf:Description ID="rearSeatLegRoom">
 <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
 <rdfs:domain rdf:resource="#PassengerVehicle"/>
 <rdfs:domain rdf:resource="#Minivan"/>
 <rdfs:range rdf:resource="http://www.w3.org/2000/03/example/classes#Number"/>
   </rdf:Description>
</rdf:RDF>
```

2.3.4 可扩展性机制

RDF 模式规范建立在由 XML 和 RDF 模型与语法提供的基础之上。它提供了一些附加的功能同时支持单个 RDF 词汇和核心 RDF 模式规范词汇的发展。

2.3.4.1 RDF 词汇的可发展性

资源描述框架设计原则之一就是使其灵活而易于扩展。这意味着将创建多种模式,这些模式的新的和改进的版本将经常在 Web 上出现。

术语学

这里所使用的短语"RDF 词汇"指的是那些随着时间发展的资源,"RDF 模式"用于指示那些在任何时间构建了特定(不变)版本的 RDF 词汇的资源。Dublin Core 词汇的每一个版本都是一个不同的 RDF 模式,并且具有一个相应的 RDF 模型和具体的语法表示。

版本和 URI 引用

RDF 使用 XML 名字空间功能[XMLNS]来识别模式,在模式中定义了属性和类。因为改变一个模式的逻辑结构就要冒着破坏另一个依赖这一模式的 RDF 模型的风险,一旦 RDF 模式改变时,则推荐名字空间 URI。

实际上,改变构成一个模式的 RDF 声明就创建了一个新的模式,新的模式名字空间应具有其自己的 URI 以避免相互混淆。因为一个 RDF 模式 URI 明确地确定了模式的一个版本,使用或者管理 RDF 的软件(如 caches)就可以安全地将 RDF 模式模型存储一段时间。RDF 模式演化问题和 XML DTD 版本管理以及 Web 资源版本的一般问题具有许多相似的特征。

因为每一个 RDF 模式具有其不变的 URI,故把这些能被用于为资源构建惟一 URI 引用的 URI 定义在一个模式中。这是通过结合资源的局部标识符和与模式名字空间相联系的 URI 来实现的。RDF 的 XML 表示使用 XML 名字空间机制来为每一个被使用的词汇术语将元素和属性与 URI 引用联系起来。

词汇间关系

RDF 模式中定义的资源本身是 Web 资源,能在其他的 RDF 模式中被描述。这一原则为 RDF 词汇演化提供了基本的机制。RDF 模式规范没有试图为表达模式间的映射提供一个完整的框架,但它确实提供了 rdfs:subClassOf 和 rdfs:subPropertyOf 属性。表达类间(subClassOf)和属性间(subPropertyOf)特殊化关系的能力提供了一个简单的机制来声明这些资源如何映射到它们的前身。

2.3.4.2 RDF 模式约束机制的演化

RDF 模式规范定义了称为"约束资源"的资源的子类。它是为增加表达 RDF 约束的新方法而提供的。资源描述框架将来的扩展可能引入新的资源,它是类 rdfs: ConnstraintResource 的实例。

当词汇术语采用未知的约束来定义时,不熟悉 rdfs: ConstraintResource 未知实例语义的 RDF 主体可能因此缺乏知识来评价约束的满意度。因为 RDF 自身不可能明确地表示这些约束资源的全部意义,关于获取一个新的 ConstraintResource 的 RDF 声明的可能不会提供足够的信息使其可供使用。比如当遇到一个先前未知的称为 RDF3: mysteryConstraint 的约束属性时,我们可能从一个模式中知道它具有一个 rdfs:Class 的范围和一个 rdfs:Property 的领域。如果单独遇到范围和领域约束,我们知道如何合法地使用 RDF3:mysteryConstraint,但是我们无法知道当以那种方式使用时,关于约束表达的特性的任何东西。

2.3.5 文档

下列属性提供 RDF 模式中简单文档和有关用户接口的注释的支持。模式的多语言文档是在语法层通过使用 XML: lang 语言标记功能来支持的。因为 RDF 模式是在 RDF 数据模型中被表示的,定义在其他名字空间的词汇可用于提供更丰富的文档。

2.3.5.1 rdfs: comment

它用于提供人可阅读的资源描述。

2.3.5.2 rdfs: label

它用于提供一个人可阅读的资源名。

2.3.6 模型和语法概念

RDF 模型和语法规范[RDFMS]引入了 RDF 的基础概念。其中许多概念在 RDF 模式中正式定义,该模式的名字空间 URI 是 http://www.w3.org/1999/02/22-rdf-syntax-ns#。另外一些进一步的概念在 RDF 模型的语法规范中引入,但没有在 RDF 模型语法模式中出现。它们正式地归入模式名字空间(比如,rdfs:Literal 和 rdfs:Resource)。

rdfs: Literal

它对应于在模型和语法规范[RDFMS]RDF 表示的正式模型中称为"Literal"的集合。原子值,如文本字符串是 RDF literal 的例子。

rdf: Statement

它对应于在模型和语法规范[RDFMS] RDF 表示的正式模型中称为"Statement"的集合。

rdf: subject

它对应于在模型和语法规范[RDFMS]RDF 表示的正式模型中称为"subject"的属性。其 rdfs: domain 是 rdf: Statement,而 rdfs: range 是 rdfs: Resource。它用于说明由一个具体化的声明所描述的资源。

rdf: predicate

它对应于在模型和语法规范[RDFMS]RDF 表示的正式模型中称为"predicate"的属性。其 rdfs:domain 是 rdf:Statement,而 rdfs:range 是 rdfs:Property。它用于识别使用在建模后的声明中的属性。

rdf: object

它对应于在模型和语法规范[RDFMS]第五节的 RDF 表示的正式模型中称为"object"的属性。其 rdfs: domain 是 rdf: Statement。这用于识别建模声明中的属性值。

rdfs: Container

该类用于表示在模型和语法规范[RDFMS]中描述的 Container 类。它是 rdfs:Class 的实例,rdfs:Resource(资源)的 rdfs:subClassOf(子类)。

rdf: Bag

它对应于在模型和语法规范[RDFMS]RDF 表示的正式模型中称为"Bag"的类。它是 rdfs:Class 的实例,rdfs:Conntainer(容器)的 rdfs:subClassOf(子类)。

rdf: Seq

它对应于在模型和语法规范[RDFMS]RDF表示的正式模型中称为“Sequence”的类。它是rdfs:Class的一个实例,rdfs:Conntainer(容器)的rdfs:subClassOf(子类)。

rdf: Alt

它对应于在模型和语法规范[RDFMS]RDF表示的正式模型中称为“Alternative”的类。它是rdfs:Class的实例,rdfs:Conntainer(容器)的rdfs:subClassOf(子类)。

rdf: ContainerMembershipProperty

该类具有成员,属性_1,属性_2,属性_3……用来指示容器成员,如同在模型和语法规范[RDFMS]所描述的一样。它是rdfs:Property(属性)的rdfs:subClassOf(子类)。

rdf: value

它对应于在模型和语法规范[RDFMS]中所描述的“Value”属性。

2.3.7 例子

本小节给出了一些使用RDF模式机制来为一些可能的应用定义类和属性的简短例子。注意这些例子中一些使用了缩写的RDF语法来表示类成员。

2.3.7.1 例1

在本例中,Person是对应于人可理解的“人的类”。Animal是一个假设定义在另一个模式中的类。所有的人是动物,所以我们说明Person是Animal的子类。人具有年龄属性。年龄的值是整数。人同样具有一个ssn(“社会安全号”)属性。其值也是整数。一个人的婚姻状况是{Single,Married,Divorced,Widowed}之一。这通过使用rdfs:range约束来达到:我们同时定义婚姻状况属性和一个类MaritalStatus(遵循惯例用小写字母开头表示属性,大写表示类)。然后我们使用rdfs: range来声明maritalStatus属性只在其具有一个值是类MaritalStatus的一个实例时才有意义。最后该模式定义了该类的一组实例。

```
<rdf:RDF xml:lang="en"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdfs:Class rdf:ID="Person">
  <rdfs:comment>The class of people. </rdfs:comment>
  <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/03/example/classes#Animal"/>
</rdfs:Class>
<rdf:Property ID="maritalStatus">
  <rdfs:range rdf:resource="#MaritalStatus"/>
  <rdfs:domain rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property ID="ssn">
  <rdfs:comment>Social Security Number</rdfs:comment>
  <rdfs:range
rdf:resource="http://www.w3.org/2000/03/example/classes#Integer"/>
```

```
        <rdfs:domain rdf:resource="#Person"/>
      </rdf:Property>
      <rdf:Property ID-"age">
        <rdfs:range
rdf:resource="http://www.w3.org/2000/03/example/classes#Integer"/>
        <rdfs:domain rdf:resource="#Person"/>
      </rdf:Property>
      <rdfs:Class rdf:ID="MaritalStatus"/>
      <MaritalStatus rdf:ID="Married"/>
      <MaritalStatus rdf:ID="Divorced"/>
      <MaritalStatus rdf:ID="Single"/>
      <MaritalStatus rdf:ID="Widowed"/>
</rdf:RDF>
```

2.3.7.2 例2

这个例子勾画了一个和可查寻 Internet 服务一起使用的 RDF 词汇的轮廓。SearchQuery 被说明为一个类。每个 SearchQuery 能同时具有一个值为 rdfs: Literal 的 queryString 和一个值为 SearchService 的 queryService。SearchService 是 InternetService(它在其他地方定义)的子类。一个 SearchQuery 具有一些 result 属性(其值为 SearchResult)。每个 SearchResult 具有一个 title(值为 rdfs: Literal),一个 rating,当然还有页面本身。

RDF 的模块性允许其他词汇与简单的模式相结合来更充分地刻画网络资源的特性。比如,Dublin Core 或者基于图书馆的分类词汇可能用于为每个 SearchService 描述主题覆盖或集合级的词汇,而一个独立的管理"查寻协议"词汇能被用于为由服务提供的 LDAP、WHOIS++ 或 Z39.50 查寻接口描述连接的详细细节。通过声明的创建,这些声明运用了来自各种领域的特定模式,RDF 使不同的专家团体为一个机器可读词汇分散的 Web 做出贡献成为可能。

```
      <rdf:RDF xml:lang="en"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
      <rdfs:Class rdf:ID="SearchQuery">
        <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </rdfs:Class>
      <rdfs:Class rdf:ID="SearchResult">
        <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </rdfs:Class>
      <rdfs:Class rdf:ID="SearchService">
        <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/03/example/classes#InternetService"/>
      </rdfs:Class>
      <rdf:Property ID="queryString">
        <rdfs:domain rdf:resource="#SearchQuery"/>
```

```
            <rdfs:range    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
        </rdf:Property>
        <rdf:Property ID="queryService">
            <rdfs:domain rdf:resource="#SearchQuery"/>
            <rdfs:range rdf:resource="#SearchService"/></rdf:Property>
        <rdf:Property ID="result">
            <rdfs:domain rdf:resource="#SearchQuery"/>
            <rdfs:range rdf:resource="#SearchResult"/>
        </rdf:Property>
        <rdf:Property ID="queryResultPage">
            <rdfs:domain rdf:resource="#SearchResult"/>
            <rdfs:range
rdf:resource="http://www.w3.org/2000/03/example/classes#WebPage"/>
        </rdf:Property>
        <rdf:Property ID="queryResultTitle">
            <rdfs:domain rdf:resource="#SearchResult"/>
            <rdfs:range    rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
        </rdf:Property>
        <rdf:Property ID="queryResultRating">
            <rdfs:domain rdf:resource="#SearchResult"/>
            <rdfs:range
rdf:resource="http://www.w3.org/2000/03/example/classes#FloatZeroToOne"/>
        </rdf:Property>
</rdf:RDF>
```

2.4 Web 数据集成的元数据解决方案

RDF 为 Web 资源描述提供了一种通用框架，它以一种机器可理解的方式表示，可以很方便地进行数据交换，RDF 提供了 Web 数据集成的元数据解决方案。通过 RDF 的帮助，Web 可以实现目前还很难实现的一系列应用，如可以更有效地发现资源，提供个性化服务，分级与过滤 Web 的内容，建立信任机制，实现智能浏览和语义 Web 等。

2.4.1 RDF 实现 Web 元数据描述与交换的机制

RDF 有两大关键技术——URI 和 XML。URI 是 Web 资源的惟一标志，它是更常用的统一资源定位符 URL 的超集，除了网页以外，它还可以标志页面上的元素、书籍和电视等资源，甚至可以标志某一个人。在 RDF 中，资源无所不在，资源的属性是资源，属性的值可以是资源，甚至一个声明也可以是资源，也就是说，所有这些都可以用 URI 标识，可以再用 RDF 来描述。那么人们在网络中如何利用 RDF 呢？XML 作为一种通用的文件格式承担了这个责任，它定义了 RDF 的表示语法，这样就可以方便的用 XML 来交换 RDF 的数据。

RDF 只定义了用于描述资源的框架，它并没有定义用哪些元数据来描述资源。这正是其高明之处。显然描述不同资源的元数据是不同的，而如果要定义一种元数据集，包括所有种类的资源，这在目前还是不现实的。不但工作量巨大，而且即使定义出这样的元数据集，能不能被大家

采纳还是个问题。因此对于图书馆这样已经用元数据描述其资源的系统,要放弃原来的元数据集,采用一种新的元数据集,其工作量是可想而知的,估计实施过程中遇到的阻力会很大。

RDF采用的是另外--种方法,即它允许任何人定义元数据来描述特定的资源,由于资源的属性不只--种,因此实际上一般是定义一个元数据集,这在RDF中被称为词汇集(Vocabulary),词汇集也是一种资源,可以用URI来惟一标识。这样在用RDF描述资源的时候,可以使用各种词汇集,只要用URI指明它们即可。当然,各种词汇集受欢迎程度可能不同,有的也许只是被定义它的人使用,有的却由于其定义的科学性为许多人所接受,如以类似图书馆卡片目录的方式来定义资源的词汇集Dublin Core,定义教育内容IMS元数据,定义个人信息的V-Card元数据等。既然词汇集是资源,当然可以用RDF来描述它的属性以及和其他词汇集间的关系,W3C为此特地提出RDF Schema来定义怎样用RDF来描述词汇集,也就是说RDF Schema是定义RDF词汇集的词汇集,但这个RDF Schema可不是随便什么人都可以定义的,它只有一个,就是W3C定义的版本。例如:

```
http://mymetadata.vocab.org/Author
---rdfs: subPropertyOf --->
http://purlorg/dc/elements/1.0/Creator
```

即表示某人自己定义的元数据Author是Dublin Core的元数据Creator的特殊形式。RDF Schema正是通过这样的方式来描述不同词汇集的元数据之间的关系,从而为元数据交换打下基础。

RDF是怎样实现Web上元数据的描述和交换呢?它使用XML语法,首先指定词汇集的URI,词汇集可以是多个,视需要而定,再使用指定的词汇集来描述资源,不同的词汇集间怎么联系呢?用RDF模式。

为了更加清楚的理解这个机制,下面来看一个用XML表达的RDF的例子:

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#
xmlns:dc="http://www.purl.org/DC/" (词汇集1的URI)
xmlns:nm="http://www.metalab.unc.edu/xml/names/"> (词汇集2的URI)

<rdf:Description about="http://www.metalah.unc.edu/xml" > (被描述资源的URI)
<dc:CREATOR parsetype="Literal">(用词汇集1的元数据CREATOR描述作者属性)
<nm:FirstName> Elliotte</nm:FirstName>(用词汇集2的元数据描述作者的姓名属性)
<nm:MiddleNmae> Rusty</nm:MiddleName>
<nm:LastName> Harold</nm:LastName>
</dc:CREATOR>
</rdf:Description>
</rdf:RDF>
```

2.4.2 RDF的特点

1.容易控制

由于RDF使用简单的资源—属性—值三元组,所以即使是数量很大的时候也比较容易控制。这个特点很重要,因为现在Web资源越来越多,如果用来描述资源的元数据格式太复杂,

势必会大大降低元数据的使用效率，其实从功能的角度来看，完全可以直接使用XML来描述资源，但XML结构比较复杂，允许复杂嵌套，不容易进行控制。采用RDF可以提高资源检索和管理的效率，从而真正发挥元数据的功用。

2.易扩展

在使用RDF描述资源的时候，词汇集和资源描述是分开的，所以很容易扩展。例如，如果要增加描述资源的属性，只需要在词汇集中增加相应元数据即可，而如果使用的是关系数据库，增加新字段可不是件容易的事情。

3.包容性

RDF允许任何人定义自己的词汇集，并可以无缝地使用多种词汇集来描述资源，以根据需要来使用，各尽其能。比如，在上个例子里描述网页资源时用Dublin Core描述其作者属性，而在描述作者的姓名时又使用了另外一个专门描述人的词汇集来描述。

4.可交换性

RDF使用XML语法，可以很容易地在网络上实现数据交换。另外，RDF Schema定义了描述词汇集的方法，可以在不同词汇集间通过指定元数据关系来实现含义理解层次上的数据交换。

5.易综合

在RDF中资源的属性是资源，属性值可以是资源，关于资源的声明也可以是资源，都可以用RDF来描述，这样就可以很容易地综合描述，以达到发现知识的目的。例如：在描述某书籍时指明其作者属性值是另一资源，可以根据描述作者的URI来获得作者的信息，如毕业院校等，从而知道这本书是某一院校的毕业生写的，于是在表面上看来没任何关系的两者间建立的联系，而这种联系往往是知识发现的前奏。

2.4.3 RDF与若干Web新技术

1.RDF与资源发现(Resource Discovery)技术

RDF采用简单的资源—属性—值三元组来描述资源。如果Web上的资源都用RDF进行描述，由于RDF采用XML语法，这样就可以很容易地实现资源的自动搜索，而不需要进行人工标引，并且可以达到很高的查全率和查准率；另外，RDF描述可以很容易进行综合，产生表面不易观察出来的信息。所有这些都将对资源发现技术产生革命性的影响。

2.RDF与个性化服务

随着Web技术的发展，个性化服务被提上日程。W3C提出的综合能力/偏好界面(CC/PP，Composite Capability/Preference Profile)推荐标准是定义网络用户以及其用来上网工具(包括硬件平台、系统软件和应用软件)性能和偏好的集合，它使用了RDF技术。可以简单认为用户及工具的能力和偏好都是用户的属性，是用户的元数据，于是可以用RDF来描述，这样就可使用同一种方法来描述Web内容和用户的能力与偏好，在用户获取信息的时候，可以通过某一种规则进行折中，以使得获取的信息符合用户的能力和偏好，为用户提供个性化服务。例如：某Web内容是用多种语言实现的，但由于翻译问题，各语种的可信度有高有低，而用户对各种语言的掌握程度也不同，这样就需要某一种规则进行折中，以让用户选择一种他可以理解的，最忠实于原文档的语种进行阅读，使用RDF描述Web内容和用户能力/偏好可大大简化这种折中的过程。

3. RDF 与 Web 信息过滤

RDF 最初提出为了配合 W3C 的 PICS(Platform for Internet Content Selection,因特网内容选择平台)规范。PICS 是由服务器向客户机传递 Web 内容等级的一种机制,比如说某一网页是否包含有色情或暴力的内容。不同机构可以按自己的价值标准将 Web 内容进行分级,这样用户就可以很容易的通过设置浏览器将某些网页过滤掉。RDF 设计的一个要求就是可以表达 PICS1.1 能表达的所有内容,以使得可以自动的将 PICS1.1 标签翻译成 RDF 的标识,而不损失任何信息,这样做的好处就是可以用 RDF 来进行数据的交换。

4. RDF 与可信任 Web(Web of Trust)

现在网络要解决的一个重要问题是建立信任机制,这个问题比较复杂,涉及到社会和技术上的许多问题。信任的一个方面是将某一声明可靠地与做出此声明的人或机构联系起来,数字签名技术则是技术上实现可信任 Web 的关键技术。W3C 提出的数字签名初步(DSig,Digital Signature Initiative)提出了一种为元数据签名的机制,以确认是谁做了这种机器可读的声明,它规定了如何由 RDF/PICS 描述签名声明,使其成为机器可识别的描述。具有数字签名的 RDF 将成为构建可信任 Web 以满足电子商务等应用需要的关键技术。

5. RDF 与智能浏览(Smart Browsing)技术

Mozilla 浏览器的 pre-Nglayout 版本和 Netscape 浏览器的 4.07 或 4.5+版本都大量采用 RDF 技术,实现了智能浏览。智能浏览即浏览器帮助浏览网页的用户提供其他与其浏览内容有关的信息,例如:如果你在 www.whitehouse.gov 上浏览白宫的网页,就有可能需要国会、国防部或者总统个人主页的 URL,而这些浏览器本身就可以提供给用户。智能浏览技术是未来浏览器发展的一个方向。

6. RDF 与语义 Web(Semantic Web)

语义 Web 是最近才提出的一个概念,即 Web 的内容不仅用来显示,更重要的是具有真正的含义,使得可以用软件工具在 Web 中漫游来处理用户提出的复杂任务。而实现语义 Web 的一个关键技术就是 RDF,因为 RDF 提供了资源的通用描述方式。语义 Web 的一个目标是突破虚拟世界的界限来控制现实世界,当我们可以用 RDF 描述电视机、电话等设备来实现对它们的协调控制的时候,想一想那是多么美好的未来!

RDF 为 Web 资源描述提供了一种通用框架,它以一种机器可理解的方式表示,可以很方便的进行数据交换,RDF 提供了 Web 数据集成的元数据解决方案。通过 RDF 的帮助,Web 可以实现目前还很难实现的一系列应用,如可以更有效的发现资源,提供个性化服务,分级与过滤 Web 的内容,建立信任机制,实现智能浏览和语义 Web 等。当然,现在的 RDF 还处于标准的制定和推广阶段,要在整个网络上都实现用 RDF 来描述资源,还有很长的一段路要走,这需要各方面的共同努力。

2.5 借助 RDF 增强 WSDL

Web 服务描述语言(WSDL)规范提供了一个基于 XML 的简单语汇表,用来描述可通过网络提供的基于 XML 的 Web 服务。这些服务本身使用简单对象访问协议(SOAP)、HTTP、SMTP 或通过其他方式进行通信,而 WSDL 为用户提供设置这些通信所需的元数据。WSDL 本身不规定如何发布或公布这种服务描述,而是将这项任务留给其他规范。“通用描述、发现和集成(UDDI)”是用来创建 Web 服务目录的一个倡议,它定义了对 WSDL 描述进行编目和

调度的一个框架,但是它刚刚出现而且相当复杂。

UDDI 是对在线合同的一个巨大促进,理应很快在分布式服务领域占据一席之地。不过,因为最初的 WSDL 部署很可能在严格封闭的系统中,而不是在开放的 Web 上,所以可能会有一种更好的替代方案。RDF 提供一些简单的方法来集成多个域中的大量元数据。

2.5.1 RDF 和 WSDL

WSDL 提供的一切本来完全可以用 RDF 序列化格式编写。IBM、Microsoft 和 Ariba 未考虑到这一点而采用了一个奇怪的选择。其他有点类似的标准,如 RDF Site Summary(RSS),说明如何将 XML 资源描述格式表示为 RDF 的 XML 序列化。这并非对所有标准都有意义。比如说,人们几乎不会期望 Scalable Vector Graphics (SVG)工作组围绕 RDF 设计 SVG。但是,当大多数信息表示简短标签和资源之间的关系时,RDF 因其不断增加的用户和工具而变得很有意义。

例如,使用 WSDL 规范编写测雪板示例,以便使用 RDF 序列化(Serialization)格式,如清单 2-1 所示。

清单 2-1:测雪板认可搜索的 WSDL 描述在用有效的 RDF 语法表示时的形式。

```
<?xml version="1.0"?>
<definitions name="EndorsementSearch"
  targetNamespace="http://namespaces.snowboard-info.com"
  xmlns:es="http://www.snowboard-info.com/EndorsementSearch.wsdl"
  xmlns:esxsd="http://schemas.snowboard-info.com/EndorsementSearch.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:w="http://schemas.xmlsoap.org/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://namespaces.snowboard-info.com"
            xmlns="http://www.w3.org/1999/XMLSchema">
      <element name="GetEndorsingBoarder">
        <complexType>
          <sequence>
            <element name="manufacturer" type="string"/>
            <element name="model" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="GetEndorsingBoarderResponse">
        <complexType>
          <all>
            <element name="endorsingBoarder" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="GetEndorsingBoarderFault">
        <complexType>
```

```
          <all>
            <element name="errorMessage" type="string"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>

  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <message w:name="GetEndorsingBoarderRequest"
             rdf:ID="GetEndorsingBoarderRequest">
      <part w:name="body" w:element="esxsd:GetEndorsingBoarder"/>
    </message>

    <message name="GetEndorsingBoarderResponse"
             rdf:ID="GetEndorsingBoarderResponse">
      <part w:name="body" w:element="esxsd:GetEndorsingBoarderResponse"/>
    </message>

    <portType w:name="GetEndorsingBoarderPortType"
              rdf:ID="GetEndorsingBoarderPortType">
      <operation w:name="GetEndorsingBoarder">
        <input rdf:resource="GetEndorsingBoarderRequest"/>
        <output rdf:resource="GetEndorsingBoarderResponse"/>
        <fault rdf:resource="GetEndorsingBoarderFault"/>
      </operation>
    </portType>

    <binding w:name="EndorsementSearchSoapBinding"
             w:type="es:GetEndorsingBoarderPortType"
             rdf:ID="EndorsementSearchSoapBinding">
      <soap:binding soap:style="document"
                    soap:transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation rdf:about="GetEndorsingBoarder">
        <soap:operation
          soap:soapAction="http://www.snowboard-info.com/EndorsementSearch"/>
        <input>
          <soap:body soap:use="literal"

soap:namespace="http://schemas.snowboard-info.com/EndorsementSearch.xsd"/>
        </input>
        <output>
          <soap:body soap:use="literal"
```

```
soap:namespace="http://schemas.snowboard-info.com/EndorsementSearch.xsd"/>
            </output>
            <fault>
              <soap:body soap:use="literal"

soap:namespace="http://schemas.snowboard-info.com/EndorsementSearch.xsd"/>
            </fault>
          </operation>
        </binding>

        <service w:name="EndorsementSearchService"
                rdf:ID="EndorsementSearchService">
          <documentation>snowboarding-info.com Endorsement Service</documentation>
          <port rdf:about="GetEndorsingBoarderPort">
            <binding rdf:resource="EndorsementSearchSoapBinding"/>
            <soap:address
soap:location="http://www.snowboard-info.com/EndorsementSearch"/>
          </port>
        </service>
      </rdf:RDF>

    </definitions>
```

正如[RDFMS]所要求的那样,这个部分包括在 rdf:RDF 元素中。如果将它输入 RDF 处理器,将获得大量信息,这些信息勾画出组成 WSDL 说明的限制条件和关系。使用可用的每个 RDF 序列化技巧将对原始 XML 结构的更改减到最少。它很好地说明了 RDF 工作组必须对 [RDFMS]进行艰苦的处理,才能使现有的 XML 格式归结到有效的 RDF 形式中。请注意,这一点也是引起极大争议的原因,因为有许多可用的技巧使得从各种语法和所生成的 RDF 抽象模型的转换具有一定的脆弱性。

这里没有将 types 元素包括在 RDF 部分。主要问题是这部分内容实际上完全超出了 WSDL 的范围。W3C 或其他 Web 标准组织可能提出一个将 XML 方案映射到 RDF 的标准,并且其他数据分类方法可能完成相同的工作,但它实际上属于完全不同的另一范畴的工作。types 描述可能仍然可通过指定 parseType="Literal" 属性完全插入 RDF 模型中,但我们同时必须面对随 parseType="Literal" 而来的所有问题;尤其是[RDFMS]明确禁止包含标记的文字之间的等价定义(例如,它可能这样规定,如果带有标记的两段文字约简到正规 XML 以后完全相同,则它们等价)。这意味着无法对以 RDF 模型存储的数据与任何其他数据进行可靠的比较。在实际应用中,这使得 parseType="Literal"毫无用处,所以不用管它。

在被转换为 RDF 的部分中,最显著的特点是在核心 WSDL 元素中添加了 rdf:ID 属性。这是 RDF 对 WSDL 作者对属性中的限定名执行的操作的处理方式:建立抽象实体之间的关系。通过使用 rdf:resource 属性就可以看到在何处生成对这些已标识资源的引用。

需要注意到的另一个特点是,所有属性前都有一个前缀。RDF 缩写允许将 namespace-resolved 属性作为特性名对待。属性值则成为语句的文字值。在清单 2-1 中大量使用了此缩写。RDF 并不严格要求特性名的名称空间,但极力建议将它作为消除这些名称的歧义的方法。如

果准备将此 WSDL 元数据放入包含其他信息的模型中，则常用的标签(如"name")很有可能与另一个特性表示(比如说，人名或组织名)发生冲突。因为 XML Namespaces 1.0 不允许应用程序将默认名称空间用做属性，所以必须明确地指定前缀来消除歧义。

在某些情况下(如消息部分)使用了匿名资源。请注意，message/part 元素有 property 属性，但没有 rdf:ID 或 rdf:about 属性。选择这样做是因为很少需要引用消息上下文之外的消息部分，而且只有此操作才需要让它为非匿名。否则，就不得不构造一个惟一的 ID(不像顶级资源那样自然)，并在 rdf:about 属性中使用 XPointer 或某个其他技巧。

可以看到 RDF 描述的任意嵌套是多么有用，它允许我们的绑定元素版本保留它原始的结构。

2.5.2 WSDL 的详细图形表示

根据 WSDL 1.0 规范，上面的示例并不是有效的 WSDL。幸运的是，从正式语法生成 RDF 非常简单，既可以通过提供 WSDL 文档中数据的同一来源的自动化处理生成，也可以使用 XSLT 生成。本节中提供的 RDF 表示仍然有用，因为它提供了用 WSDL 编码的元数据的一个具体 RDF 模型。这样，使用从 WSDL 中抽取的任何特定 RDF 语法来构造相同的模型就是一件简单的事了。

WSDL 的 RDF 模型的一大优点是它直接导致 WSDL 模型所指定内容的方便可视化。通过将元数据映射到 RDF，已将它套进一种形式化中，[RDFMS]为此形式化提出了一种有用的表示作为定向图。如要看实际操作，可访问 Dan Brickley 的 RDF 可视化工具网站(http://www.dfki.uni-kl.de/frodo/RDFSViz/)，并在 URL 文本框中输入下面的 URL：

http://www-4.ibm.com/software/developer/library/ws-rdf/endorse.rdf。

请快速浏览一下生成的图形。如果使用可视化工具的 GIF 输出，则有些困难，所以如果有诸如 Adobe SVG Plug-in 之类的查看器，建议使用 SVG 输出。此外，也可以从这个观测器站点生成 2 维虚拟现实标记语言 (VRML)。请注意，已为匿名资源分配了一个生成的统一资源标识符 (URI)，这个标记符对于 RDF 工具很有用，尽管它与[RDFMS]中给出的图形样例有所不同，在[RDFMS]中匿名资源用空椭圆表示。例如，分配到的 URI 包含在第一个 message 元素中的匿名端口资源是：

http://www-4.ibm.com/software/developer/library/ws-rdf/endorse.rdf#genid3。

另请注意，显式绘制的 rdf:type 属性，在清单 2-1 中是使用分类节点隐式绘制的，而没有使用 rdf:description 元素。

2.5.3 用于 WSDL 的 RDF 模式

Dan Brickley 的 RDF 可视化工具的图中揭示的类型特性暗示了一种用于 WSDL 的 RDF 模式，该方案可转换为一种 RDF 模型的形式框架。WSDL 规范仍然不提供任何 RDF 模式，但在清单 2-2 中对其中的原因提出了最恰当的建议。它并不是 WSDL 的一个完整 RDF 模式，它只包含清单 2-1 中的示例的一个子集，而此子集又只使用了 WSDL 的一个子集，但这可望成为进一步工作的基础。

清单 2-2：一个可能的 WSDL RDF 模式

```
    <? xml version="1.0"? >
<!--
```

```
Note: to be used to frame the listing 1 example, it would require
a base uri of "http://schemas.xmlsoap.org/wsdl", or an RDF processor
that can manipulate base-URIs (such as 4RDF)
-->
<rdf:RDF
   xml:lang="en"
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#' >

         <!--                          Class message                     -->
   <rdfs:Class ID="message">
     <rdfs:comment>An individual transmission in XML-based
communication.</rdfs:comment>
     <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
   </rdfs:Class>
         <!--                          Class part                     -->
   <rdfs:Class ID="part">
     <rdfs:comment>A distinct section of a message.</rdfs:comment>
     <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
   </rdfs:Class>
   <rdf:Property ID="element">
    <! More accurately constrained to be a Qname -->
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    <rdfs:domain rdf:resource="part"/>
   </rdf:Property>

         <!--                          Class portType                     -->
   <rdfs:Class ID="portType">
     <rdfs:comment>A resolution of individual messages into a set of
operations.</rdfs:comment>
     <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
   </rdfs:Class>
   <rdf:Property ID="operation">
    <rdfs:range rdf:resource="operation"/>
    <rdfs:domain rdf:resource="portType"/>
   </rdf:Property>

         <!--                          Class operation                     -->
   <rdfs:Class ID="operation">
     <rdfs:comment>A set of messages that comprise a single logical request and optional
response.</rdfs:comment>
```

```
    <rdfs:subClassOf
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
   </rdfs:Class>
   <rdf:Property ID="input">
    <rdfs:range rdf:resource="message"/>
    <rdfs:domain rdf:resource="operation"/>
   </rdf:Property>
   <rdf:Property ID="output">
    <rdfs:range rdf:resource="message"/>
    <rdfs:domain rdf:resource="operation"/>
   </rdf:Property>
   <rdf:Property ID="fault">
    <rdfs:range rdf:resource="message"/>
    <rdfs:domain rdf:resource="operation"/>
   </rdf:Property>

      <!--                    Common                    -->
   <rdf:Property ID="name">
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
    <rdfs:domain rdf:resource="message"/>
    <rdfs:domain rdf:resource="part"/>
    <rdfs:domain rdf:resource="portType"/>
    <rdfs:domain rdf:resource="operation"/>
   </rdf:Property>
 </rdf:RDF>
```

应注意的第一点是，大小写与 RDF 模式规范中使用的约定不一致。这是为了避免改变 WSDL 所用的元素类型名。除此之外，它还是循环的方案。只要 WSDL 对描述元素之间的内部关系有相应的限制，就使用 domain- 和 range- 约束。举例来说，因为 WSDL 输入元素中的消息属性的值必须是消息名，所以 RDF 模式使用清单 2-3 中代码段的等价范围约束。

清单 2-3：RDF 模式的范围约束

```
<rdf:Property ID="input">
      <rdfs:range rdf:resource="message">
      <rdfs:domain rdf:resource="operation">
</rdf:Property>
```

上面的范围约束确保输入特性只能在操作元素中使用，这也是 WSDL 的一个约定。

请注意，其中许多约束并不是它们所能达到的最强约束。例如，元素属性指明消息部分使用的类型部分中的特定元素。该特性的值实际上应该是一个有效的特性名字。例如，我们不能将字符串“!! 42xyz??”设置为该特性的值，因为它甚至不是一个有效的 XML 元素类型名，但是因为我们将 rdfs:Literal 用作该特性的范围，所以 RDF 处理器将会容忍这种错误。遗憾的是，RDF 模式几乎没有提供对数据分类的支持，这个任务留给了以后的版本，那个版本将能够利用 XML 模式小组的工作。

第3章 XML的语义表示和处理方法——XML与本体技术的结合

在这一章中，首先从语言的角度对XML在语义表示方面的优势和不足进行分析，归纳出XML在语义互操作方面的需求，从而引入本体的概念。在此基础上，深入分析并总结本体的构造方法论，基于本体构造领域信息语义模型的理论，并结合XML标准介绍当前具有代表意义的本体表示和交换语言XOL和OIL，同时对RDFS用于本体表示的方法进行探讨，最后介绍一种将描述逻辑表示的本体转化为DTD的方法。

3.1 XML与语义

XML如何帮助在语义层次进行信息编码？是否能够有所帮助？对这些问题，许多接触过XML的用户都称XML是一种"语义标记"，称赞XML具有通过标记清晰地表达语义的能力。我们的确能够理解这些用户的观点：XML文档将专有的、过程的和隐式的标记进行了标准化。在文本编辑器中将XML文档与其他直观表示的标记语言进行比较，如Word文档，PS格式的文档等，用户可能会立即得出XML文档比其他的文本文件具有更多"意义"的结论。标记本身作为元数据的一种形式，能够向用户解释什么是文档的组成要素以及这些信息对象如何构成更大的一致的信息单元。

在SGML出现时，就存在对描述性标记的基本原则的争论，即描述性标记能否以一种有意义的、自描述的方式对SGML/XML文档中开始和结束处双重界定的信息对象进行描述。通过领域专家精心选择的元素类型名称、属性类型名称和属性值，描述性标记确实能够帮助人类对标记之间的信息进行预测。但是，描述性标记本身作为一种计算机信息交换机制存在许多局限。XML的<BOOKTITLE>与HTML的<I>相比，看起来具有清晰明了的语义，但是对于XML处理器，<BOOKTITLE>、<I>具有相同的语义。

与它的父辈元语言SGML一样，XML不具备支持语义完整性约束声明的机制，甚至即使在XML DTD中非形式化地声明了信息对象的语义，XML处理器也无法验证信息对象的语义。XML处理器之所以自身不能够理解文档对象的语义，是因为XML标记语言没有预先定义应用层处理语义。因此，XML在形式上统一了语法，而不是统一了语义的表示。

实际上，XML语法被设计用于表示一种串行化的编码，对复杂对象语义建模的表达能力非常有限，XML难以表示问题域中对象的概念化模型。XML表示关系的基本构造是"容器"(表示层次关系)，"邻接"，"并发"，"属性"和"非透明的引用(Reference)"。这些结构确实有利于串行化，却不是用户将问题域中对象抽象建模的理想选择。所有的基本关系语义都必须以这些语法结构进行表达，并且XML处理器仍然无法识别它们的意义。

因此，为了充分开发XML支持数据交换和互操作计算的能力，必须将XML处理和某种计算机可处理的语义定义结合起来。应用领域的XML文档不仅应该能够进行语法检查，而且能够进行语义的有效性检查。这些都意味着需要共享本体(Ontology)或对象语义的公共集合存在。

3.2 XML 语义互操作问题

第一代的 Web 使人们能够通过浏览器对信息进行访问，是一种非自动化的有限集成的 Web。第二代 Web 使计算机和人类都能够访问信息和服务，是一种自动化的，支持智能检索的，由多种主体构成的 Web。下一代的 Web 将把人类、计算机和各种机器连结起来，是一种支持过程控制、提供智能环境的新一代 Web。XML 被认为是新一代 Web 互操作性的基础之一，能够以一种中立于应用和厂商的形式进行信息交换（如图 3.1 所示）。

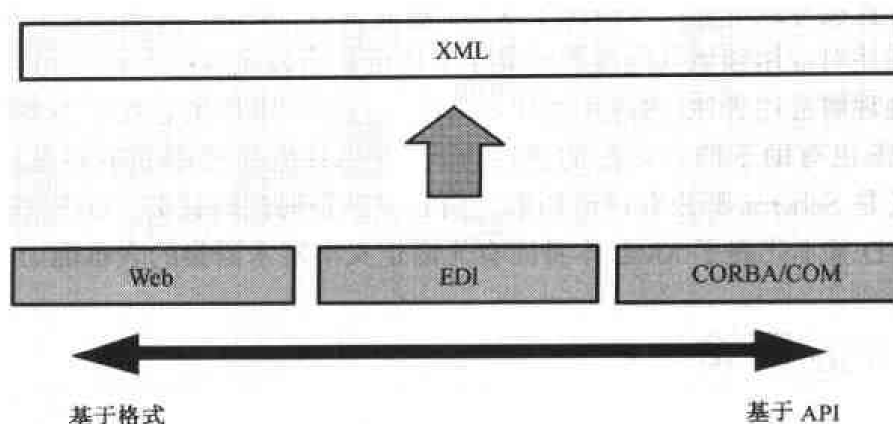


图 3.1　XML 作为互操作的基础之一

XML 在形式上统一了信息交换的语法，但在语义互操作性方面仍然存在一些不足，Web 智能化的复杂性类似于“智能主体”的复杂性，归纳起来主要有：

信息采用不同的词汇、分类方法和模式；

隐式体现的语义（隐含在词汇中或来源于背景和模式的语义）；

背景、意图和处理之间缺少必要的分离；

没有明确声明的嵌人式的规则；

Web 文档模型难以满足应用互操作的需求；

维护不同版本、不同标准的信息之间的语义一致性；

……

电子商务需要丰富的信息交换，特定的 XML 格式确实能够改善单个信息交换任务的情况，但是特定的格式意味着要错过利用大量共享处理软件的机会，因为 XML 本身的数据模型过于底层。元数据联盟（Meta-Data Coalition）等机构试图通过“开发一个基于 XML 的开放的基础架构，促使电子商务所有的参加者在全球能够以一种互操作的，安全的，一致的方式使用电子商务的信息”。不过，所有这些方法都必须将不同类型商务中的不同术语体系进行定义，进而定义不同商务领域中信息的语义。具体地讲，出于以下考虑，以 XML 为基础的互操作需要显式地定义信息的语义：

(1) 简单的 XML 表示方法是否能够支持大型的、松散耦合的贸易团体之间的自动商业行为。

(2) 在 Web 内容的语义足够支持根据需求建立特定的贸易关系以前，还有许多重大的技术挑战需要克服。

(3) 初步的 XML DTD 标准化本身不足以支持建立复杂有效的商务，也难以推动其发展。

(4) 参与者之间保持行为一致的需求。

(5) 在谈判、仲裁等复杂交互过程中保持行为一致的需要。

(6) 在柔性、动态的环境中,参与者采用多种不同的平台和标准保持行为一致的需要。

(7) 各种"智能主体"只有在它们交换的内容以及它们提供的服务中携带充分的显式说明信息时,它们的工作才是有意义的。

显式定义语义信息的基础概念已经在分布式问题求解(DPS)、智能主体和知识共享等研究领域建立起来了。XML 为这些概念和方法以一种既符合 Web 标准,又符合商业需求的形式重新应用奠定了基础。

这些考虑都直接导致本体技术应用于 XML 研究领域:即为了共享信息和知识,需要一个不同应用之间描述对应用领域共同理解的共享术语的集合以及这些术语之间的关系。本体将帮助应用部分地理解应用领域,为应用增加灵活性。这样的术语集合被称为本体。

XML 的视图也有助于约束词汇的使用方式,并为数据结构添加某些意义,但是无论是 XML 的 DTD 还是 Schema 都没有讨论词汇之间的关系和词汇的意义。DTD 可以看做是非常初级的本体,DTD 基本代表了 XML 本身固有元语定义丰富术语集的表示能力。

3.3 Ontology 理论

自从 20 世纪 90 年代以来,围绕本体召开了为数众多的专题研讨会。来自哲学、知识获取和表示、计划、过程管理、数据库视图集成、自然语言理解和企业建模等领域的,历史上相互分离的不同领域的研究人员,从各自的角度出发共同探讨本体问题的核心。1998 年这一领域的第一个主题会议"信息系统中形式本体论国际会议"(ICFOIS1998)召开,同时伴随着相关研究成果数量和质量的增加,标志着这一领域的研究日趋走向成熟。

3.3.1 本体是什么

以大写"O"开头的 Ontology 是指哲学领域的本体论这一概念。在哲学中 Ontology 是一种存在的系统化解释,用于描述事物的本质。本体论的概念和方法被计算机领域采用,用于知识表示、知识共享和重用。以小写"o"开头的 ontology 是计算机领域广泛使用的概念,翻译为本体。直观地讲,本体是一个实体,是对某领域应用本体论的方法分析、建模的结果,即把现实世界中的某个领域抽象为一组概念及概念之间的关系。

为了澄清在知识工程领域本体的概念,Guarino 和 Giaretta(1995)针对流行的本体 7 种不同概念的解释进行了深入分析,给出了目前基本上得到了 AI 领域认同的概念界定,即 ontology 是一个概念化(Conceptualization)的显式(Explicit)说明或表示。

要明确上述概念界定的含义,首先要明确"概念化"的含义。概念化从广义上讲是指世界观,是指对某个领域的思维方法。它可以被看做是"限制实体(Reality)某一部分结构的非形式化规则的集合"(Guarino, 1997)。它也可以被典型地理解并表示为"概念的集合(例如实体,属性,过程)以及它们的定义和相互间的联系"(Uschold&Gruninger,1996)。概念化可以是隐式的,例如存在于人的头脑中,或嵌入在软件中。

上面对本体的定义并不是最终的标准定义,但是却符合绝大多数普通的标准用法,对知识工程更具有指导意义。如上的定义明确地要求本体是显式的,以区别于概念化。除此以外,该定义对本体没有更多的限制。

在知识工程领域，本体总是以某种方式与表示共享知识的知识库设计相关联。本体理论与任意逻辑理论(或知识库)的不同之处在于理论的语义部分。因为本体理论所有的公理在基础概念化的任何可能世界中都必须是成立的。实际上，如果本体和知识库使用相同的语言定义，它们之间并没有清晰的界限，本体库的设计与实现可以借鉴普通知识库的方法。

3.3.2 本体的基本内容

在知识工程领域，所谓“存在”就是可以被表示。我们可以通过定义描述性的术语，来描述应用的本体。在人工智能领域，本体是一个工程性的人造物(Engineering Artifact)，它由一个用于描述某种现实情况的特定术语集和一组显式定义的公理集组成，这组公理用于描述上述术语的内涵，通常采用一阶逻辑的形式。可以采用多种不同的形式表示本体，但是一般都包含一个术语的词汇表和词汇意义的某些说明。在最简单的情况下，本体只描述由包含关系关联起来的概念层次。复杂的本体包括概念的定义和概念相互之间的关系以及概念和概念之间关系所满足的公理，它们共同地在领域(Domain)上施加一个结构，限制对术语可能的解释和应用。

实际上本体是许多主体协定的对某个领域共享理解的表示。这种协定有助于精确内容意义、高效地通信，同时又促使系统的交互式操作、重用和共享等一系列的性能得以提高。图 3.2 是一个采用框架逻辑(Framework Logic)表示的本体实例，图中表示的内容为某领域研究人员本体的一部分，是对研究人员(Person)和出版物(Publication)的概念以及研究人员的合作关系(CooperatesWith)，研究人员与出版物之间相互关系公理的定义。

```
Object[ ].
    Person :: Object.
    publication :: Object
Person[
    firstName =>>STRING;
    lastName =>>STRING;
    eMail =>> STRING;
...
publication=>>publication].
publication[
    author=>>Person;
    title=>>STRING
    year=>>NUMBETR;
    abstract=>>STRING}.
```

```
FORALL Person1,Person2
    Person1:Researcher[cooperatesWith->>Person2]
<-
Person2:Researcher[cooperatesWith->>Person1].

FORALL Person1,publication1
    publicaton1:publication[author->>Person1]
<->
Person1:Person[publication ->> Publication1].
```

图 3.2　框架逻辑的本体实例

3.3.3 本体的研究现状

人工智能领域对本体的研究主要包括以下三个方面的内容：

本体论工程。研究和开发本体的内容，包括两个方面，一是研究如何创建特定领域的本体，二是研究通用本体的创建方法。

本体的表示、转换和集成。研究用于表示各种本体的知识表示系统,提供形式化方法和工具,促进本体的共享和重用,提供不同本体的比较框架,研究不同本体的转换和集成方法,提供不同本体间互操作的手段。

本体论的应用。主要研究以特定领域或通用本体为基础的应用。

对特定领域本体的研究和开发目前已经涉及了许多领域,包括企业本体,医学概念本体,酶催化生物医学本体,电子商务供应链本体等。

比较著名的企业本体研究工作包括爱丁堡大学企业项目(Enterprise Project)和多伦多大学的虚拟企业(Virtual Project)项目。

通用本体论最著名的研究项目是 CYC 工程,通过近二十年的开发,该工程已经建立了一个包罗万象的巨大知识库。通用本体研究的另一个分支是 Chandrasekaran 等的关于任务和问题求解方法本体的研究工作。这项工作主要研究可共享问题的求解方法,与领域无关的推理方法。

典型的关于知识表示系统的研究工作包括 Stanford 大学知识系统实验室(Knowledge Systems Laboratory,KSL)从事的关于知识本身的本体研究,研究包括知识的本质特征和基本属性。在该项研究中,提出了 KIF(Knowledge Interchange Format)知识描述语言,该项工作的研究重点是语言对知识的描述能力,为应用和主体之间的知识交换和共享提供了通用的格式。KIF 采用一阶谓词逻辑,支持对象、函数和关系的定义,允许元级知识和非单调推理规则的表示,具有描述性的语义。目前众多其他知识的表示形式都能够转换为 KIF 形式,KIF 被认为是知识共享的桥梁。以 KIF 为基础,KSL 正式建立了 KSL 本体服务器,通过 Internet 供全球研究人员使用,并提出了开放知识互联(OKBC)的概念。为了满足 XML 本体表示的需求,基于 OKBC 的语义相继推出了 XOL(XML-based ontology-exchange)等 Internet 知识表示语言。

3.3.4 本体在互操作方面的应用

从目前发表涉及本体研究与应用成果的文献中可以发现,总体上说应用本体都是为了使系统获得某种方式的共享和重用。某些文献将创建本体看做是构造知识库的一种途径,另外一些将本体看做是知识库的一部分,此外还有将本体视为与应用相关的交互工具。

根据已有的文献,按照应用领域的不同可以大致将本体的应用划分为三大类,如图 3.3 所示。

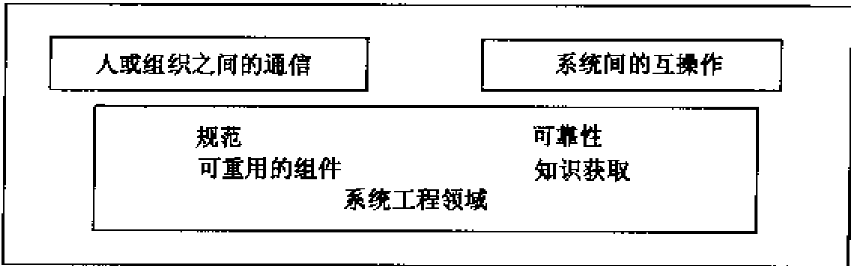


图 3.3 本体应用的三个主要领域

达成对概念的共识对人或组织间交流的促进作用是显而易见的,因此本体很自然地应用于人或组织之间的通信。

通过使用本体作为应用领域的概念模型,不同的建模方法、算法、语言和软件工具之间能够共享相同的背景知识,实现系统间的互操作,具体过程如图 3.4 所示。

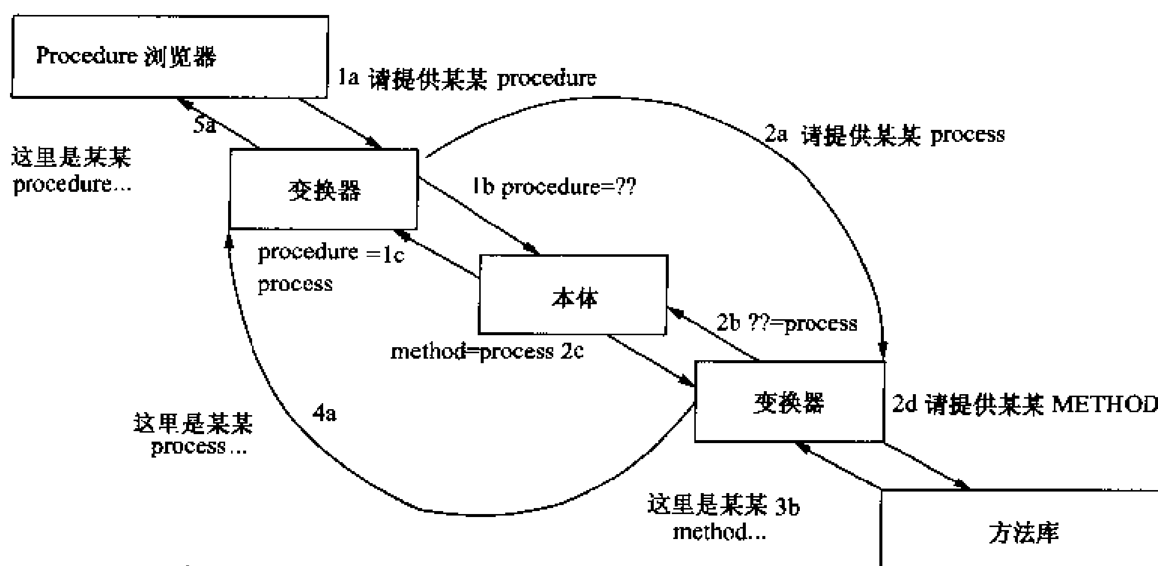


图 3.4　应用本体实现系统互操作示例

本体在信息表达方面与语法，词汇表和语言之间的关系如图 3.5 所示。述行语(Performatives)和言语行为(Speech Acts)表明了言语的目的和意图。语法和句法规定了言语的拼写和结构。本体定义了言语应用的背景及其在该背景下对言语意义的解释。缺少明确定义的本体，主体间就不可能实现对信息语义的理解和共享。

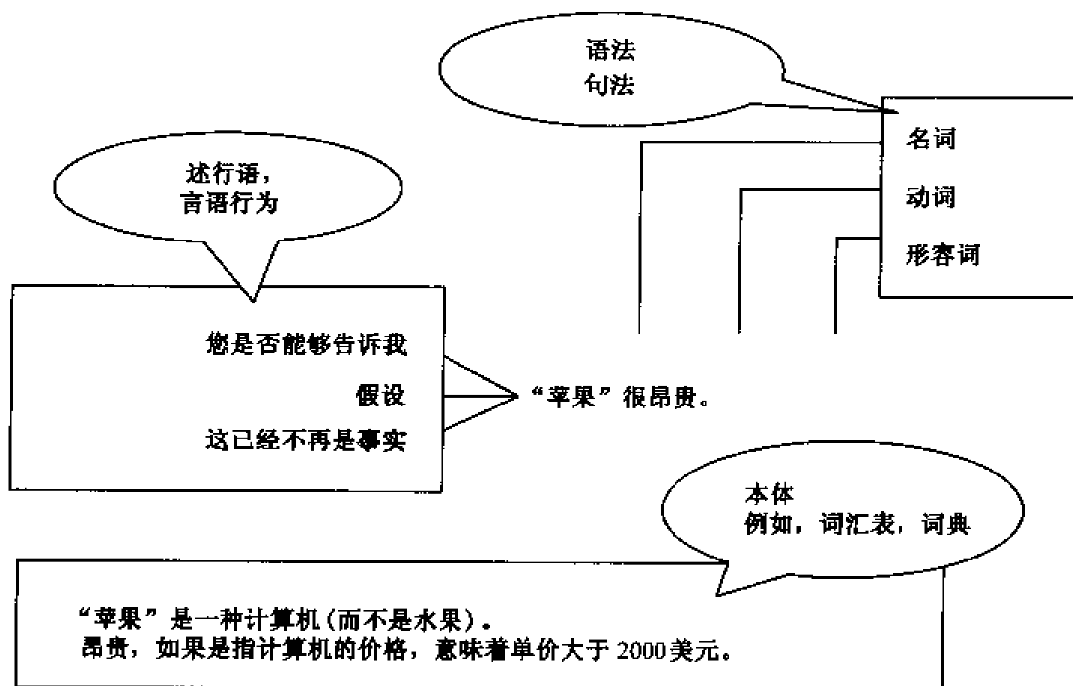


图 3.5　本体在信息表达方面的作用示意图

将本体技术应用于系统工程领域，在以下四个方面有助于提高系统的性能：

可重用性：本体是领域中重要的实体、属性、处理过程和它们之间关系形式化编码的基础。这种形式化的表示可以是软件系统中可重用的或共享的组成部分。

知识获取：在构造基于知识系统时，使用已经存在的本体作为引导知识获取的起点和基础，能够有效地提高系统的速度和可靠性。

可靠性：形式化的表示，使自动的一致性检查成为可能，从而使软件更加可靠。

规范：本体能够辅助一个 IT 系统识别处理需求，定义各种规范。

3.4 Ontology 的开发和应用

Fernández 与 Gómez-Pérez(1996)总结了在化学领域中构造本体的经验，归纳了作者在工程实践中符合本体开发过程的行为集，在进化快速原型方法基础上建立了本体开发的生命周期，提出了一种本体的结构化开发方法以及开发过程中多种中间结果的表示技术。Gruber (1993 年)从工程的角度总结了形式化本体的开发过程，提出了指导这种开发过程的一组标准，将这些标准应用于工程数学和书目数据两个应用领域的示例研究。

企业本体(Enterprise Ontology, EO)是大型企业建模基础设施的一个重要组成部分，它覆盖了企业建模所有的核心概念。Uschold(1998)等全面总结了构造本体的初衷，在定义本体中的各项内容的过程，将它们转换为形式化定义的经验、教训，对开发大型本体中可能遇到的普遍性的问题以及将非形式化本体转换为形式化本体时几类典型问题进行了甄别，并提出了一些解决途径。

Guarino(2000 年)讨论了基于本体建模工程方法论。这种方法建立于从哲学研究中借鉴过来的一些重要的分析概念，包括：同一性(Identity)、整体性(Unity)、严格性(Rigidity)和依赖性(Dependence)，并将它们用于本体建模的分析过程中，为领域的本体分析引入了基本的规则。

上述的这些工作从各自的实践经验中出发，描绘出了本体建模工程的过程、方法和步骤的轮廓，同时也不可避免地存在一些片面性和不足。主要反映在两个方面：缺少对本体建模工程标准、指导原则进行落实的可操作性强的方法；另一方面针对建模工程中某些特定阶段具有良好应用价值的分析、建模方法，又缺少将它们与整个建模工程集成的研究。此外，许多开发者直接用自身熟悉的形式化语言对本体进行编码，将对领域概念化模型的假设和设计标准隐含在编码中。这种开发方法既违背了本体开发的宗旨，又阻碍了本体的共享和重用。其结果就是本体建模更像是一门“艺术”，而不是一项工程。

本章探讨将哲学领域基础的本体论概念引入到知识工程本体建模中的方法，从而有效地解决现有建模方法缺少形式化分析工具、难以记录分析过程的问题。

3.4.1 本体的开发方法

本体建模方法属于基于知识系统(Knowledge-based Systems, KBS)的开发，但普通开发 KBS 的方法不能完全适用于本体的建模。这是因为通常在开发 KBS 时，知识工程师很难定义系统在应用领域中具体、完整的工作方式，所以一般的 KBS 系统采用进化的原型系统方法进行开发。开发本体的目的是用于人类、计算机对知识的共享和重用，它应该是相对稳定的，独立于具体的应用。在本体建模的起点就必须详细说明模型中涵盖的概念、实例、关系和公理等实体，至少是初步认定描述这些实体的绝大部分词汇。

综合现有系统的开发过程，用本体建模的生命周期对本体建模的方法、步骤和设计标准进行了有机的集成。本体建模的生命周期从总体上可划分为规约制定、概念化和实现三个主要的阶段，如图 3.6。评测、集成和文档编写贯穿于开发的全过程，知识获取主要集中于前两个阶段。

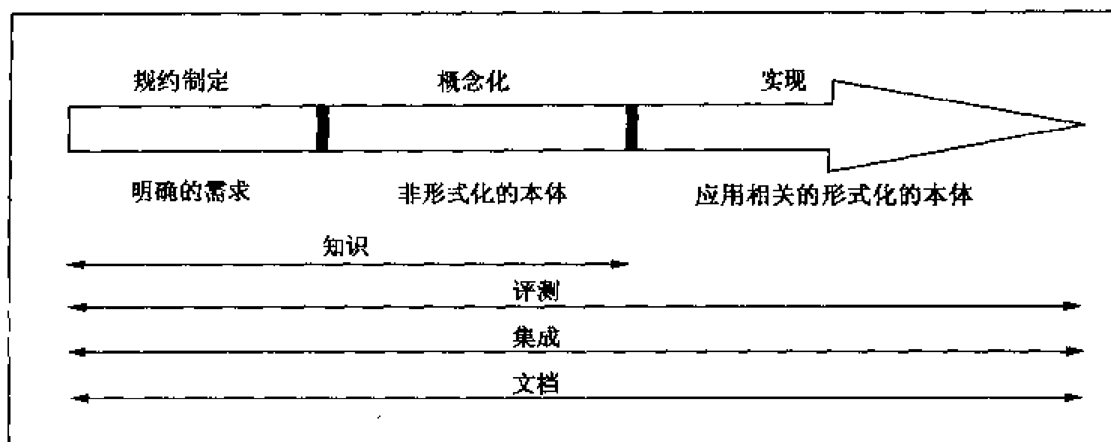


图 3.6　本体建模的生命周期

规约制定阶段：主要的任务是以文档的形式详细说明开发本体的目的和领域，明确为什么要开发本体，预期本体的用途和最终的用户。这一阶段的中间结果是本体开发目的和领域的详细说明书，说明书的形式根据预期的最终用户而定。例如 EO 和(TOVE，The Toronto Virtual Enterprise) 都是企业建模领域的本体，由于面向的最终用户不同，EO 和 TOVE 分别采用自然语言和形式化的语言制定规约。

概念化阶段：主要任务是统一开发人员对领域概念化模型的认识，并以一种明确的方式详细记录概念化的模型。这一阶段是本体开发的重点，将在下一节进行详细的讨论。

实现阶段：主要任务是使用形式化的语言对概念化阶段产生的领域概念化的模型进行编码，使计算机能够对概念化模型进行理解和处理。这种形式化语言被称为目标语言，目标语言的选择与具体的应用相关，可参考 Speel(1996)对多种目标语言的研究和比较，在此不做进一步的讨论。

由于系统开发的阶段性，每一阶段都会产生相应的中间结果，都有产生中间结果的标准和假设；每一阶段都有集成领域内已有本体和其他成果的可能；每一阶段所做的工作都需要记录。所以在整个开发过程中都需要评价、集成和编写文档。

规约制定阶段和实现阶段的工作与具体的应用紧密相关，并且可以直接借鉴知识工程中相关的理论和方法。概念化阶段是本体建模的核心阶段，现有方法的不足之处主要集中于这一阶段，因此这里将重点研究本体建模的概念化阶段。

3.4.1.1　领域的概念化

领域概念化阶段要将开发者对领域概念化分析过程和概念化的结果以适当的形式明确地记录下来。如何记录概念化的结果已经有了比较完善的解决方案。概念化分析过程的记录涉及到概念化过程中对领域的假设，设计者的准则等难以明确表达的内容，往往被本体的开发者忽略。Guarino 等人在领域概念化分析过程中引入 Identity(同一性)，Unity(完整性)，Rigidity(严格性)，Dependence(依赖性)四个本体的基本性质，从分类关系涉及的参数性质入手，为分类关系特别是包含关系提供了形式化的分析工具。使用这些形式化工具能够将分析的过程和分析过程中的各种假设清晰地记录在分析结果中，而且对领域概念分类的结果按照不同的假设形成不同的层次结构。

形式化分析工具

形式化分析工具主要是通过定义元属性(Meta-property),在分析中引人本体论的基本概念,再使用元属性对属性相对于本体论基本概念的行为特性进行形式化分析。为了行文方便,在新概念第一次出现的地方,概念名称正文可能使用英文其后加中文的注释,概念再次出现时,视情况使用英文或中文;对标准的逻辑、模态逻辑的概念和规则不作解释。元属性用大写字母表示,在字母前加"+"、"-"或"~"分别表示承载,半承载和排斥元属性。例如,ϕM 表示属性 ϕ 具有元属性 M。

关于 Rigidity 的定义

定义 1:元属性 rigid property(严格属性)是指属性的所有实例都必须满足的属性。

定义 2:元属性 non-rigid property(半严格属性)是指对于属性的某些实例不必要的属性。

定义 3:元属性 anti-rigid property(非严格属性)是指对于属性的所有实例都不必要的属性。

严格属性、半严格属性和非严格属性等元属性分别用 +R, -R 和 ~R 标记。例如 PERSON 在每一个可能的世界中都是人,因此可以做如下 PERSOM^{+R}标记。STUDENT 的任何一个实例都不必总是学生,因此可以做如下 $\text{STUDENT}^{\sim R}$标记。

关于 Identity 的定义

在哲学著作中,任意属性 ϕ 的同一性条件(Identity Condition,IC)通常被定义为满足下列公式的适当的关系 ρ:

$$\phi(x)\wedge\phi(y)\rightarrow(\rho(x,y)\leftrightarrow x=y) \tag{1}$$

这种定义在建模工程中存在以下的问题。首先是具有 IC 的属性是承载 IC 还是提供 IC 的问题。Non-rigid 属性只能承载 IC,而不能提供 IC。例如 $\text{STUDENT}^{\sim R}$只能承载从 PERSON^{+R}继承而来的 IC。第二个问题是关系 ρ 的性质问题:如何判断 ρ 能够成为 IC;如何用 ρ 来解释同时(Synchronic)同一性和历时(Diachronic)同一性之间的区别。第三根据(1)决定一个属性是否承载了 IC 可能非常困难。因为关系 ρ 对于同一性条件要同时满足充分和必要条件。因此对公式(1)作如下的改进:

定义 4:IC 是满足如下公式(2)或(3)的公式 Γ,假设 E 是表示实际存在的谓词,并且排除平凡错的情况:

$$E(x,t)\wedge\phi(x,t)\wedge E(y,t)\wedge\phi(y,t)\wedge x=y\rightarrow\Gamma(x,y,t,t) \tag{2}$$

$$E(x,t)\wedge\phi(x,t)\wedge E(y,t)\wedge\phi(y,t)\wedge\Gamma(x,y,t,t)\rightarrow x=y \tag{3}$$

满足(2)的 Γ 为必要的 IC,满足(3)的 Γ 为充分的 IC。

定义 5:任何属性 ϕ 承载 IC,当且仅当它被能够提供 IC 的属性包含(包括自身能够提供 IC 的属性)。

定义 6:属性 ϕ 中提供 IC,当且仅当

Ⅰ ϕ 是严格的

Ⅱ ϕ 承载了必要或充分的 IC

Ⅲ 相同的 IC 不能被所有包含 ϕ 的属性承载。这意味着如果 ϕ 从多个不同的属性继承多个 IC 时,ϕ 仍然能够提供 IC。

定义 7:任何承载 IC 的属性 ϕ 被称为是可以划分种类的(Sortal)。

元属性 +I 标记承载 IC 的属性,反之使用元属性 -I 标记。元属性 +O 标记提供 IC 的属性,反之使用元属性 -O 标记。根据以上定义,+O 蕴涵 +I 和 +R 显然成立。

关于 Unity 的定义

从直观上讲完整性是指某事物的每一个组成部分都以某种方式与事物的其他组成部分相连接，同时在这种方式下不与任何其他的事物连接。这种完整性的理解被形式化定义为：

定义 8：事物 x 在关系 ω 下是一个整体，当且仅当 ω 为等价关系，x 的所有部分被 ω 连接在一起，同时 ω 不连接其他的任何事物。

定义 9：属性 φ 承载了整体性条件(Unity Condition，UC)，当且仅当存在一个单一的等价关系 ω，φ 的所有实例在关系 ω 下作为一个整体。

根据关系 ω 的本体论性质，ω 可以被理解为是一种"广义的连接"。对于具体的实体，可以分辨出整体性的三个主要种类，它们分别是拓扑整体性、形态学整体性和功能整体性。

定义 10：当属性 φ 的每个实例都不能构成一个整体时，属性 φ 具有非整体性(Anti-unity)。

元属性 + U 标记承载 UC 的属性，反之使用元属性 - U 标记。元属性 ~U 标记属性具有非整体性。显然，~U 蕴涵 - U。

关于 Dependence 的定义

Guarino 等人重点研究应用于属性性质的本体论依赖概念。在此定义的依赖概念建立于 Simon"概念的依赖(Notional Dependence)"定义基础之上，是对更加广义的外部属性的近似的定义。

定义 11：对于属性 φ 和属性 ψ，如果对于 φ 的所有实例 x，ψ 的某些实例必须存在，并且这些实例不是 x 的一部分或构成要素，则称属性 φ 外部依赖于属性。

$$\forall x \Box (\phi(x) \rightarrow \exists \varphi(y) \wedge \neg P(y,x) \wedge \neg C(y,x)) \quad (4)$$

P，C 代表部分和要素关系。

约束和假设

以上元属性为分类关系施加了一些具有操作性的约束，尤其是针对包含关系。如果 φ 和 ψ 是两个属性，那么以下的约束成立：

ϕ^{-R}不能包含 φ^{+R}。 (5)

ϕ^{+I}不能包含 φ^{+I}。 (6)

ϕ^{+U}不能包含 φ^{-U}。 (7)

$\phi^{\sim U}$不能包含 φ^{+U}。 (8)

ϕ^{+D}不能包含 φ^{-D}。 (9)

具有不相容 IC 和 UC 的属性不相交。 (10)

以上的约束条件能够从元属性定义中直接得到。

最后对同一性作如下的假设：

(1) 种类的个体化：每个领域的元素一定实例化了某些承载 IC(+ I)的属性。

(2) 种类的可扩充性：如果两个实体相同，一定存在承载这种同一性条件的属性，并且这两个实体是这种属性的实例。

3.4.1.2 形式化分析方法

我们首先探讨以上元属性不同组合方案的意义。不同的组合方案对应于根据本体论基本概念对属性划分的基本种类。这些属性种类作为使用元属性对给定属性特征进行分析的参考，同时用于对分析结构的合法性检查。I，O，D 分别取布尔值，R 取 + R，~ R，- R 3 值，共有 24 种潜能的组合。但是由于 + O → + I，+ O → + R，所以实际的组合种类为 14 种，如表 3.1

所示。共有 8 种基本属性种类，这 8 种基本属性种类有可以划分为可分类的(Sortal)属性和不可分类的(Non-sortal)属性两大类。此外我们将承载 + R 元属性的属性统称为骨干属性。

范畴(Category)类属性通常是本体中最高层次的属性，将领域划分为不同的片断。

表 3.1 属性的本体论分类

<table>
<tr><td rowspan="2">+O</td><td rowspan="2">+I</td><td rowspan="2">+R</td><td>+D</td><td rowspan="2">类　型</td><td rowspan="8">可分类的</td></tr>
<tr><td>+D</td></tr>
<tr><td rowspan="2">+O</td><td rowspan="2">+I</td><td rowspan="2">+R</td><td>+D</td><td rowspan="2">伪类型</td></tr>
<tr><td>+D</td></tr>
<tr><td>+O</td><td>+I</td><td>~R</td><td>+D</td><td>具体角色</td></tr>
<tr><td>-O</td><td>+I</td><td>~R</td><td>-D</td><td>阶段可分类的</td></tr>
<tr><td rowspan="2">-O</td><td rowspan="2">+I</td><td rowspan="2">-R</td><td>+D</td><td rowspan="2">混合的</td></tr>
<tr><td>-D</td></tr>
<tr><td rowspan="2">-O</td><td rowspan="2">-I</td><td rowspan="2">+R</td><td>+D</td><td rowspan="2">范畴</td><td rowspan="6">不可分类的</td></tr>
<tr><td>-D</td></tr>
<tr><td>-O</td><td>-I</td><td>~R</td><td>+D</td><td>形式化角色</td></tr>
<tr><td rowspan="3">-O</td><td rowspan="3">-I</td><td>~R</td><td>-D</td><td rowspan="3">特性</td></tr>
<tr><td rowspan="2">-R</td><td>+D</td></tr>
<tr><td>-D</td></tr>
<tr><td rowspan="3">+O</td><td>-I</td><td></td><td rowspan="3"></td><td rowspan="3">不合理的</td><td rowspan="3"></td></tr>
<tr><td rowspan="2">+I</td><td>~R</td></tr>
<tr><td>-R</td></tr>
</table>

类型(Type)类属性是本体中最重要的一类属性。因为领域中的任何元属性至少是一种类型的实例，所以它也是对领域进行描述时使用最多的一类属性。

伪类型(Quasi-type)类属性通常是建立在多种属性组合基础上的属性类型，用于对领域进行高层次的组织。

形式化角色(Formal Role)类属性通常用于描述实体参与事件的哪些部分，将多个实体间的特殊关系具体化。

具体角色(Material Role)类属性表示具体到特定实体种类的形式化角色。

阶段可分类的(Phased Sortal)属性是指不能提供全局 IC，但是属性的实例在特定的时间阶段里能够提供局部 IC 的属性。

特性(Attribution)类属性用于表示品性或性质的取值。

混合(Mixin)类属性通过合并 + R 和 ~ R 属性对特殊实体进行描述。

3.4.1.3 应用实例

下面以逐步澄清一个杂乱的术语分类的实例进一步说明上述方法。术语分类是从 WordNet，Pangloss 和 Mikrokosmos 等现有的本体中抽取的 is-a 关系(见图 3.7)，具体的处理过程如

下：

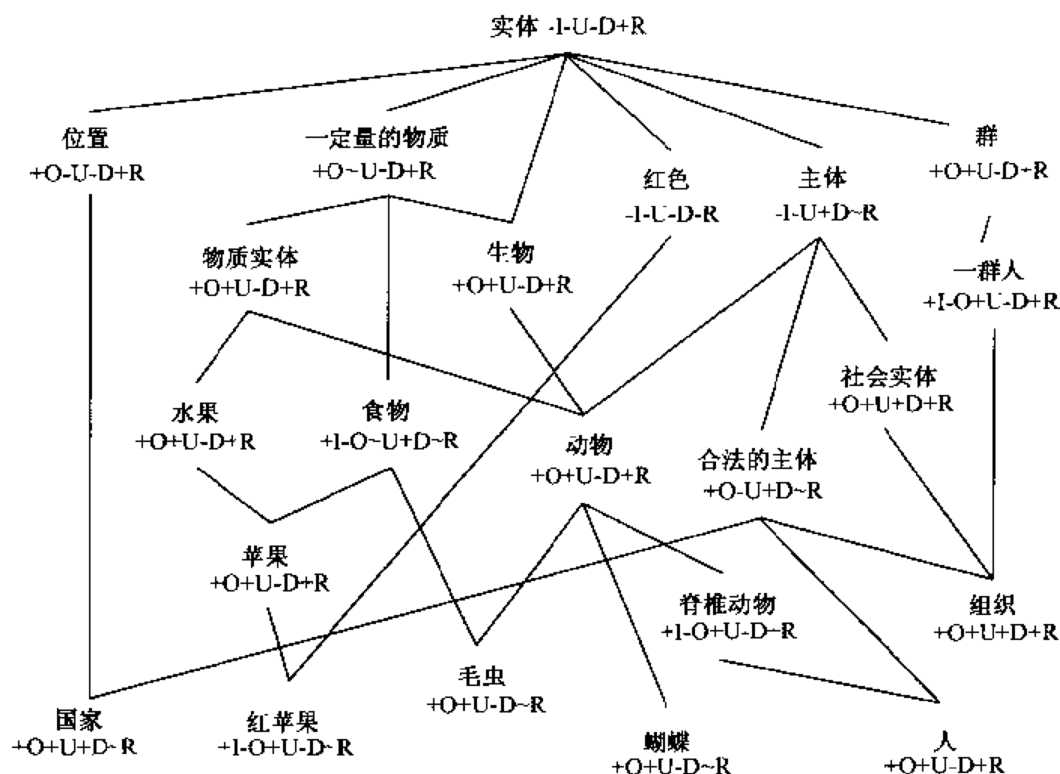


图 3.7 一个杂乱的词汇分类

根据相应的元属性澄清术语分类中关于每个属性的本体论假设。词汇的元属性是根据词汇最普通的意义确定的。

下面将根据元属性和它们的约束验证这些假设的一致性。我们认为这里所作的假设是有道理的，但是这些假设未必完全正确。我们的目的是探究在特定的本体中这些假设的结论。

“位置”既可以指空间区域也可以指时间区域，它们可以是连续的，也可以是离散的。出于这些考虑，我们不认为它具有整体性，因此具有元属性～U。“主体”从直观上认为事物只有在参与某种行为时，事物才被认为是“主体”，因此具有元属性～R。物理对象是一个隔离的物质实体，因此“苹果”具有元属性＋U。“脊椎动物”被认为是有脊椎的动物，而不是具有脊骨的任意的动物。“社会实体”被认为是具有某些“社会同一性”的具有生命的人的复数。“合法的主体”被认为是牵扯进法律契约的“主体”。最后，将“国家”简单地认为是处于一个行政状态的地理区域。

检查每个元属性集的一致性。大家已经看到我们的元属性彼此之间不是独立的，只能够为＋R属性定义元属性＋O。在实例中很自然地认为“国家”同时具有＋O和～R，但是与定义7矛盾。我们进一步分析对“国家”的假设，我们将“行政区域和地理区域”和并为一个，但是它们的同一性属性却截然不同。因此将“国家”划分为“地理区域”和“国家”，它们拥有自己的同一性和完整性。“国家”被分类在“社会实体”和“合法的主体”之下（见图3.8）。

删除所有不具有＋R元属性的属性。这是识别骨干分类的第一步，结果如图3.9所示。

根据元属性性质的逻辑结果，检查施加在每个分类关系上的约束。“物质实体”和“生物”到“一定量的物质”的两个连结由于违反了完整性约束而被删除。“动物”和“物质实体”之间的连接，由于不一致的IC而被删除：当一个动物死亡后它就不存在了，但是物理对象还存在。同

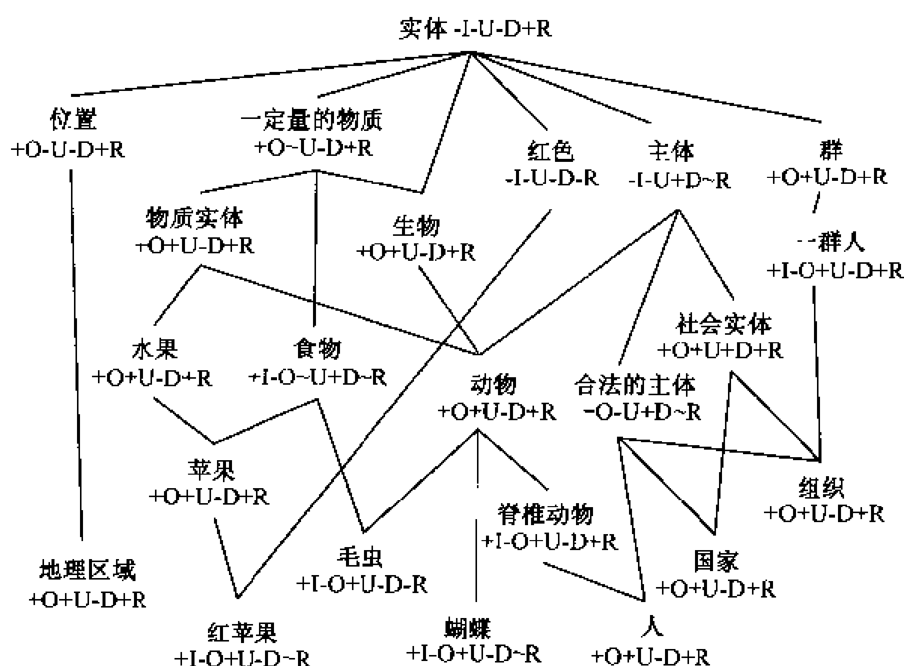


图 3.8 修正“国家”的假设

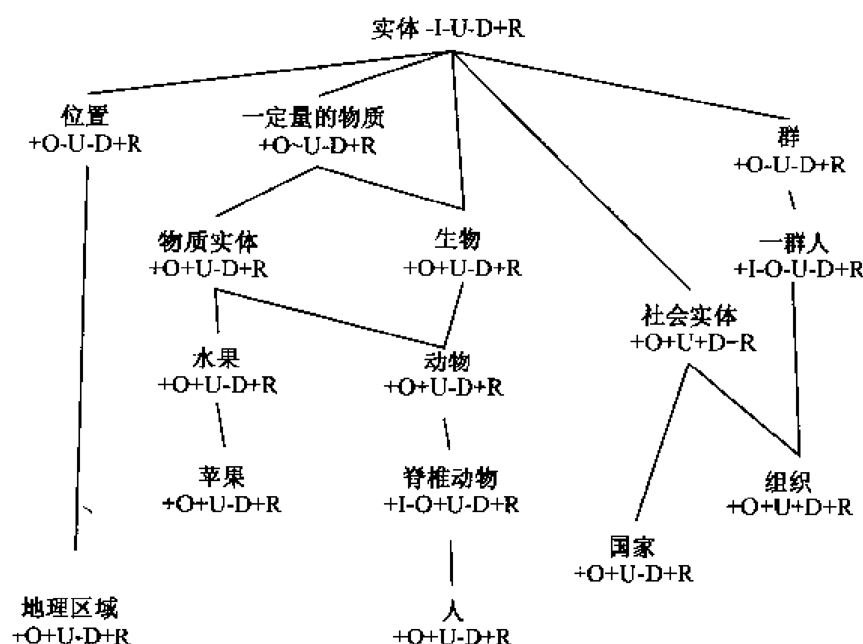


图 3.9 限定具有 + R 的元素

样，“组织”不仅是人类的群体，因为相同的人群能够组成不同的组织。这些操作的结果反映在图 3.10 中。

添加其他的属性，检查可能的约束冲突。在本例中，由于～R 不能够包含 + R，“主体”和“合法的主体”不允许包含“人”，“组织”和“国家”。从表面看起来这导致了信息的丢失：我们曾经认为“人”是“合法的主体”，现在这条假设如何表示？这一问题的解释是因为本例中主要是表示 is-a 关系，尽管“人”可以是“合法的主体”，但是并不是所有的“人”都是“合法的主体”。“食物”的分析相似，结果如图 3.11 所示。

检查遗漏的概念。我们已经看到不相容的同一性意味着不相交的分类，但是反之却不一

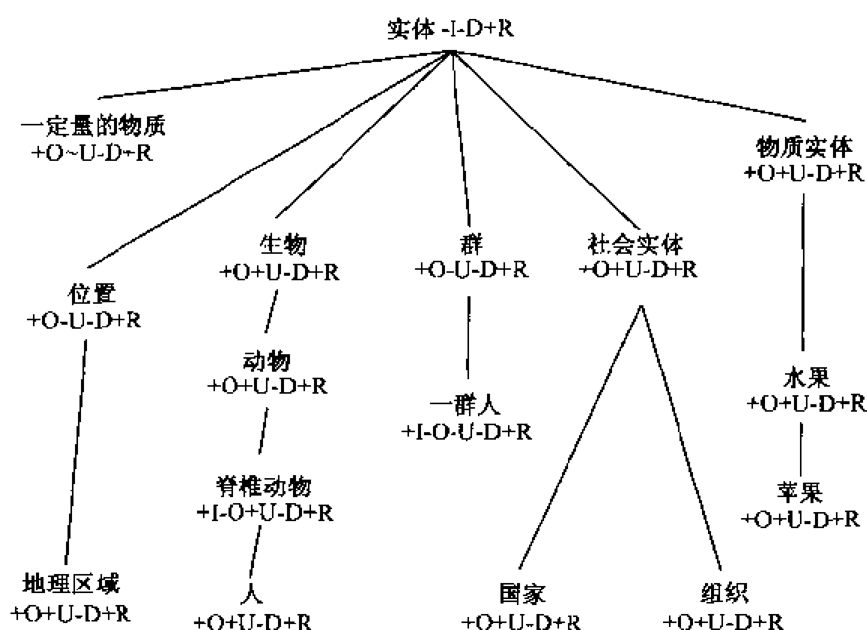


图 3.10 初步的骨干分类

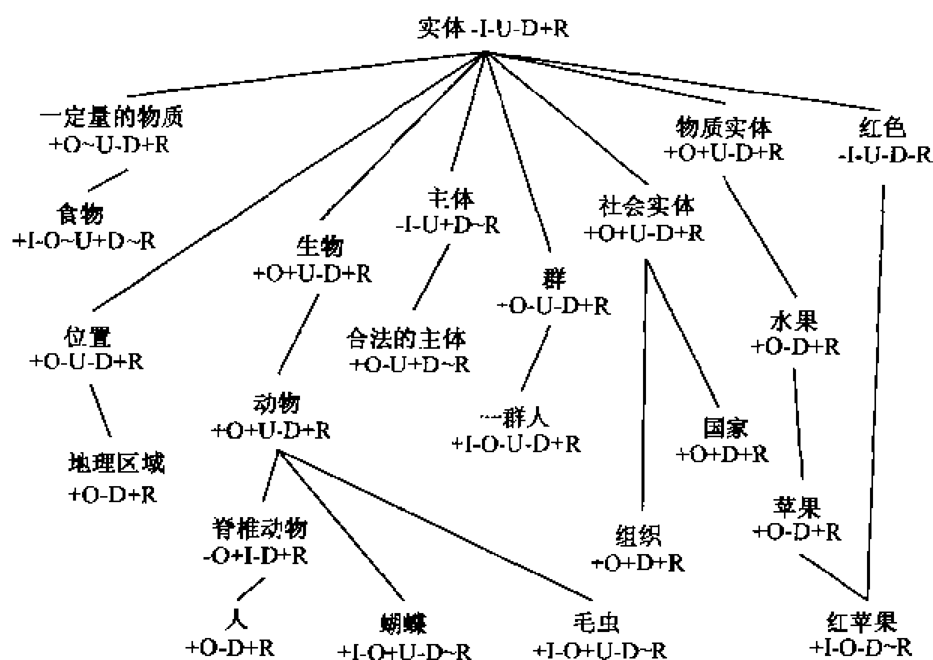


图 3.11 添加其他属性，骨干分类使用加粗表示

定正确。例如在我们的实例中，“蝴蝶”和“毛虫”与“人”不相交，尽管它们之间不存在同一性方面的不相容。同时，我们知道“毛虫”和“蝴蝶”并非是不相交的，因为同一个昆虫在初期是毛虫，后期是蝴蝶。为此，我们有理由添加一个包含它们的概念“鳞翅类”。它提供了不同于“人”的自身的 IC(图 3.12)。

通过介绍这一本体设计方法的基本步骤，我们可以看到这一方法为知识工程师提供了两个好处：

根据施加在 is-a 关系上的语义约束，澄清了分类关系，识别了骨干分类结构。

这一方法使分析者将本体论的约定清晰化，澄清了概念应用的预期意义，生成了更为合理

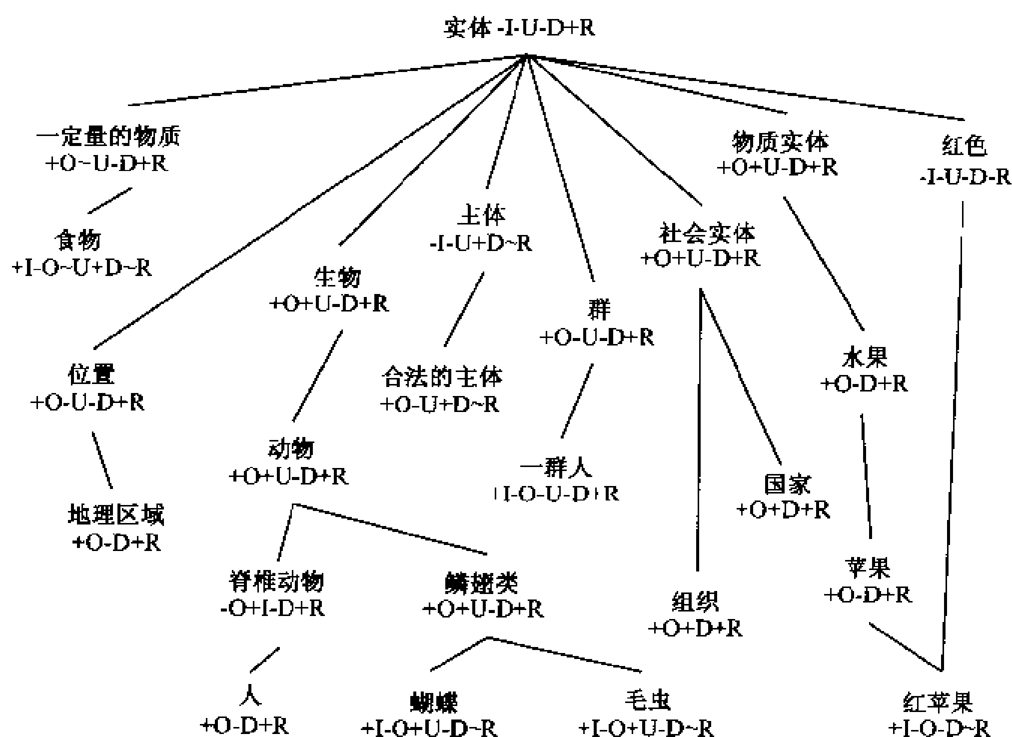


图 3.12　最终的分类

的本体。

3.4.2　本体与 XML 结合的基本方法

在上一节介绍了基于本体的语义信息处理的三个根基之一(图 3.13)。在这一节将主要介绍基于 XML 语法的语义表示和处理方法。下面重点介绍在 XML 文档中表示和处理本体的方法。

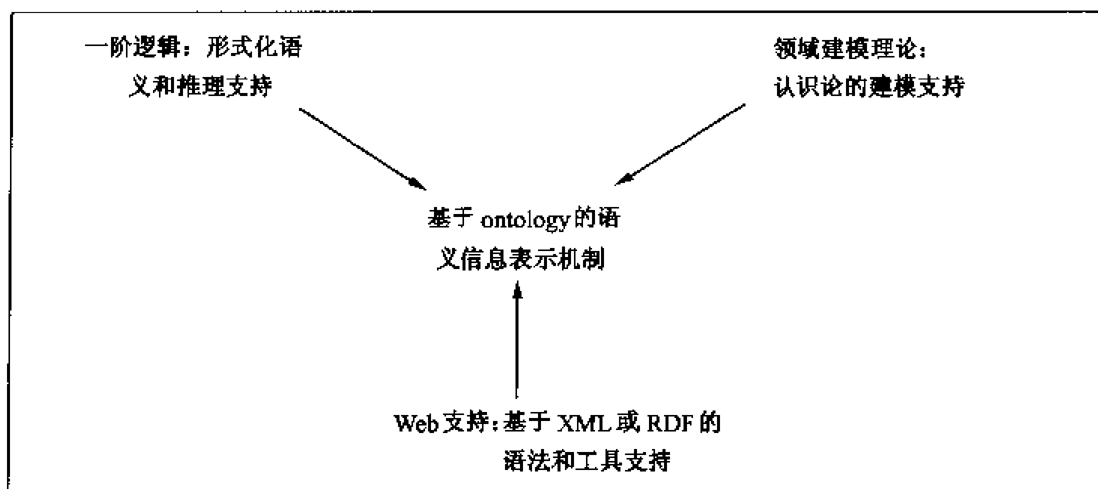


图 3.13　基于本体的语义信息表示的三个根基

XML 协议簇被设计用于在 WWW 上更加形式化地表示和处理信息，并得到广泛的推广。XML 文档是显式结构化的文本文档，能够被应用程序方便地访问，为构造 Web 信息的语义表示奠定了良好的语法基础。同时，XML 只是定义文档结构的描述语言，并不具有描述语义信

息的能力。文档结构能够表示有限的语义,但是从文档结构中反映出来的语义信息只适用于特定目的的应用。所以我们需要在其基础上,为 XML 添加语义信息的表达能力。

XML 文档将文档的内容、结构和表现分开定义。XML 文档结构的定义手段通常有两种,一种是 DTD(Document Type Definition,文件类型定义),一种是 XML 模式方法。XML 模式本身就是采用 XML 表达的,能够被主流的 XML 工具处理,作为 XML 指定的 DTD 的继承者,具有更强的表达能力。它能够为 XML 的属性或元素定义数据类型,它的元素能够继承其他元素的属性和内容模型,因此具有比 DTD 更为紧凑的结构,更加适合于体现相应本体的内容,所以各研究项目都将其作为今后研究的方向。但是,目前完全支持 XML 模式的工具非常少,并且处于试验阶段,因此目前主要采用 DTD 进行语义表示研究。将语义信息和文档的结构相关联是比较合理的切入点,将 XML 文档的结构与本体相关联,再利用 XML 文档结构与 XML 内容之间已有的关系将 XML 的内容与本体相关联,提供信息的语义表达能力。

构造本体的语言一般要具有丰富而直观的表达能力,具有一阶逻辑的推理能力。当前一般是采用描述逻辑(Description Logics,DL)和框架逻辑(Frame Logic,FL)形式化地定义本体。DL 的长处在于它结合了包含检查、自动分类和实例识别算法。但是它的主要不足之处在于它限定的表示形式,只容许术语知识的说明,不能够推导出新的术语知识。

FL 开发目的在于提出一种面向对象和基于框架语言的陈述逻辑。它提供了对象标识、复杂对象、继承、多态类、方法和封装,并将这些特征集成在一个基于逻辑的框架内。FL 语法是高阶的,FL 将这种高阶语法与模型理论语义和可靠、完整的基于归结的证明过程相结合。但是在 FL 的具体实现中存在良好模型语义所要求的层次化问题。由于 XML 文档在局部的分层是事先不可确定的,需要针对层次化方面加强的语法。FL 灵活的语法使这方面的工作非常困难。我们将介绍 XOL 和 OIL,分别说明采用 XML 表示的描述逻辑和框架逻辑定义的本体。同时,我们还将介绍 RDF 如何作为一种本体描述语言,最后介绍一种将描述逻辑定义的本体转换为 DTD 的方法。

3.4.2.1 基于 XML 的本体交换语言 XOL:Xml-based Ontology exchange Language

XOL 是一种基于 XML 语法和 OKBC 语义的本体交换语言。OKBC 采用框架逻辑中类(Classes),槽(Slots),面(Facets)和个体(Individuals)等概念进行知识建模。OKBC 中定义的原子数据类型为:

- integer
- floating point number
- boolean
- string
- name of class

在 OKBC 中,类是实体(Entities)的集合。如果类的实体是类本身,那么称这种类为元类(Metaclass)。如果实体不是类,那么我们称这种实体为个体。在 OKBC 中预定义了以下的类:THING, CLASS, INDIVIDUAL, SYMBOL, NUMBER, STRING, INTEGER。其中类 THING 是类层次的根(图 3.14)。

使用 instance-of 关系表示实体与类的成员关系,例如 instance-of(Ind,Cls)为真,如果实体 Ind 属于类 Cls。type-of 是它的逆反关系。subclass-of 关系表示类的层次,它只适用于类。subclass-of(Csub,Csuper)为真,当且仅当 Csub 的所有实体都是 Csuper 的实体。它的逆反关系

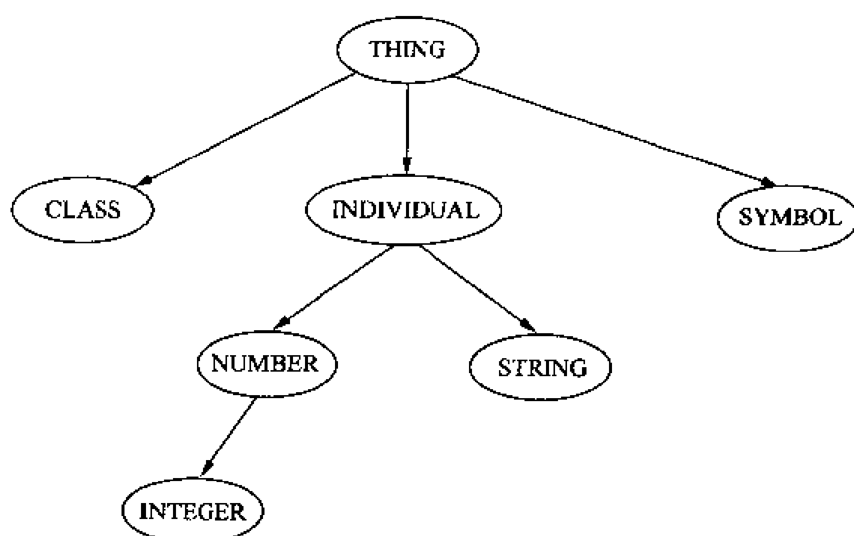


图 3.14　OKBC 中预定义类的层次

是 superclass-of。

每个类的实体有与自身相关的槽的集合。每个槽有与自身相关的槽值的集合。

我们可以认为槽是实体和自身槽值之间维持的一个二元关系。槽自身有与之相关的自身的面，面依次有自身的面值。例如我们可以认为书“Birdy”是类 BOOK 的实体，具有 AUTHOR, ISBN, PUBLISHER 等槽。ISBN 的槽值为 0679734120，我们可以为槽 ISBN 定义面 CARDINALITY，其值为“1”。自身的槽描述特定个体的属性，属于相同类的其他实体可以拥有不同的槽。

有时，我们定义的槽所描述的属性可能会对类中的每个个体都有意义。这样的槽被称为模板槽(Template Slot)，同样我们可以定义模板面(Template Facet)。模板槽能够被继承，例如每个类都有它父类的模板槽。类，个体，槽，面由框架统一表示。

OKBC 定义了一个标准的槽：文档(Documentation)，该槽用于为类/实体提供文档说明。槽值的类型为字符串类型。

OKBC 定义了一些标准的面，它们是：

- VALUE-TYPE
- CARDINALITY
- MINIMUM-CARDINALITY / MAXIMUM-CARDINALITY
- NUMERIC-MINIMUM / NUMERIC-MAXIMUM
- COLLECTION-TYPE

在此，我们将解释 VALUE-TYPE 面的语义，其他面的详细解释请参见 OKBC 的定义。VALUE-TYPE 为槽的取值定义了一个类型约束。例如，如果槽 S 的 VALUE-TYPE 面为类型 T，那么如果某个实体的槽 S 的取值一定为类型 T。

有时，有必要定义槽的属性(面)，这些面对所有使用槽的框架(类或实体)都成立。要实现这种功能，可以在槽上定义槽，例如：槽框架(Slot Frame)的槽。这种类型的槽中最重要的就是 DOMAIN(域)，它定义能够使用这个槽的类。换句话说，如果 S 是一个槽，C 是一个类，DOMAIN 关系在 S 和 C 之间成立，那么：

(1) 如果 Ind 是槽 S 上被赋值的实体，Ind 一定属于类 C。

(2) 如果 Csub 是槽 S 上被赋值的类(例如,S 是 Csub 的模板槽),Csub 或者是 C 的子类,或者是 C 本身。

面 DOMAIN 的默认值为 THING。如果我们不显式为槽定义 DOMAIN,那么每个类都使用默认值。

其他为槽框架定义的槽有:

- SLOT-VALUE-TYPE
- SLOT-CARDINALITY
- SLOT-MINIMUM-CARDINALITY / SLOT-MAXIMUM-CARDINALITY…
- SLOT-NUMERIC-MINIMUM / SLOT-NUMERIC-MAXIMUM
- SLOT-COLLECTION-TYPE

我们详细介绍使用 VALUE-TYPE 面的 SLOT-VALUE-TYPE 槽,其他槽可以使用相同的解释方法。如果槽 S 的 SLOT-VALUE-TYPE 值的类型为类 T,并且 C 是一个类,那么 S 和 C 之间的 DOMAIN 关系成立,类 C 的所有实体将 T 作为面 VALUE-TYPE 的值。

例如,我们有两个类,BOOK1 和 BOOK2 都具有 ISBN 模板槽,但是 BOOK1 定义 VALUE-TYPE 槽为 string 类型,BOOK2 的 VALUE-TYPE 槽为 number 类型。那么 BOOK1 的所有实体只能够为 ISBN 赋字符串类型的值,而 BOOK2 只能赋数字类型。另一方面,如果我们将 ISBN 定义为槽,那么 BOOK1 和 BOOK2 都在 ISBN 槽的 DOMAIN 中。此时,如果我们将 ISBN 的 SLOT-VALUE-TYPE 定义为 string 类型,那么我们能够确保 BOOK1 和 BOOK2 的所有实体的 ISBN 槽只能取 string 类型的值。

到目前为止,我们已经介绍了 OKBC 定义的类、槽和面。下面,我们将介绍 XOL 如何在 XML 中描述这些概念。每个 XML 文档表示的本体都将符合 XOL DTD 或者 XSchema。每个 XOL 文档由以下部分组成:

- module section
- class section
- slot section
- individual section

XOL 文档包含且只包含一个模块,它标识了被描述的本体。所有其他的部分都包含在模块部分中。首先浏览一下 XOL DTD 中的模块表示实例:

```
<module>
    <name>Media ontology</name>
    <kb-type> some famous KBS</kb-type>
    <version>1.0</version>
    <documentation>Ontology for online books and videos </documentation>
</module>
```

根据 DTD,模块部分的根元素可以是下列元素之一,可以认为它们是同义词:module, ontology, kb, database, dataset。

根元素必须拥有一个名字(Name)元素作为本体的名字,其他的元素是可选的,kb-type 定义本体知识库的来源和依据,版本(Version)和文档(Documentation)定义关于本体的附加信息。

接下来的是类部分。在这里我们通过如下的实例,介绍本体中的类,介绍它们的关系和模板槽。

```
<class>
    <name>MEDIA</name>
    <documentation>The class of all media types</documentation>
    <subclass-of>THING</subclass-of>
</class>
<class>
    <name>BOOK</name>
    <documentation>The class of all books</documentation>
    <subclass-of> MEDIA </subclass-of>
</class>
<class>
    <name>VIDEO</name>
    <documentation>The class of all videos</documentation>
    <subclass-of> MEDIA </subclass-of>
</class>
<class>
    <name>DVD</name>
    <documentation>The class of all DVDs</documentation>
    <subclass-of> MEDIA </subclass-of>
</class>
<class>
    <name>ORGANISATION</name>
    <documentation>The class of all organizations</documentation>
    <subclass-of> THING </subclass-of>
</class>
<class>
    <name>COMPANY</name>
    <documentation>The class of all companies</documentation>
    <subclass-of> ORGANISATION</subclass-of>
</class>
```

下面介绍包括槽框架在内的槽定义。每个槽框架根据<slot>标签 type 属性或者描述一个模板槽,或者描述一个自身的槽。我们为 BOOK 类定义如下的槽:AUTHOR, ISBN, TITLE, PUBLISHER。

```
<slot type="own">
    <name>AUTHOR</name>
    <documentation>The name of the book/video author </documentation>
    <domain>BOOK</domain>
    <domain>VIDEO</domain>
    <slot-value-type>STRING</slot-value-type>
    <slot-minimum-cardinality>1</slot-minimum-cardinality >
```

```
    <slot-maximum-cardinality>3</slot-maximum-cardinality >
</slot>
```

注意我们允许 BOOK 和 VIDEO 的个体拥有 author,但是不允许 DVD 拥有。

```
<slot type="own">
    <name>ISBN</name>
    <documentation>The ISBN of the book</documentation>
    <domain>BOOK</domain>
    <slot-value-type>STRING</slot-value-type>
    <slot-cardinality>1</slot-cardinality >
</slot>
<slot type="own">
    <name>PUBLISHER</name>
    <documentation>The publisher of the media</documentation>
    <domain>MEDIA</domain>
    <slot-value-type>COMPANY</slot-value-type>
    <slot-cardinality>1</slot-cardinality >
</slot>
```

接下来的是个体部分:

```
<individual>
    <name>Knopf</name>
    <documentation>Knopf publishing company</documentation>
    <instance-of>COMPANY</instance-of>
</individual>
<individual>
    <name>Birdy</name>
    <documentation>cool book</documentation>
    <instance-of>BOOK</instance-of>
    <slot-values>
        <name>AUTHOR</name>
        <value>William Wharton</value>
        </slot-values>
        <slot-values>
        <name>ISBN</name>
        <value>0679734120</value>
    </slot-values>
    <slot-values>
        <name>PUBLISHER</name>
        <value>Knopf</value>
    </slot-values>
</individual>
```

XOL 文档必须遵守的规则如下:

(1) 在同一篇 XOL 文档中 classes/individuals/slots 的名字必须惟一。

(2) 父类定义必须先于子类定义。

(3) 类定义必须先于它们的实体。

(4) instance-of 和 subclass-of 提供的标识符必须是类的名字。

(5) slot-values 和 slot 的约束与上面相同。

(6) 槽只能够被它们域中的类和实体使用。

(7) 槽的值要与 SLOT-VALUE-TYPE 定义一致。

(8) 类定义要先于它们的槽定义。

同时,面的值应该有意义。但是在 XOL DTD 中不能强制限定这一规则,XML 模式中可以限定。

此外,在这里我们将指出 XOL DTD 和 XOL 模式中的一些不足:

(1) 尽管 OKBC 声明对于面 COLLECTION-TYPE 和槽 SLOT-COLLECTION-TYPE 合法的取值为 set, list 和 bag 三者之一,而 XOL 定义相应的标记为 PCDATA 类型的值。这意味着可以取任何字符串值,这将导致应用处理 XOL 表示的本体时产生混乱。

(2) XOL 没有区分 OKBC 中加以区分的 primitive 和 non-primitive 类型的类。

(3) DTD 要求类部分先于槽部分,这可能导致模板槽的不一致。类可能为一个没有定义的槽赋值。不过,这对于自身类不构成问题,因为槽部分先于个体部分。

3.4.2.2 本体交换语言 OIL:Ontology Interchange Language (version 1.0)

在设计 XOL 时,就已经考虑本体交换语言的设计,但是 OIL 不仅是 XOL 的延伸。实际上这两种语言中存在着许多不同。OIL 的设计者来自三个领域,它们分别是:描述逻辑的形式化语义,基于框架逻辑的建模元语和基于 XML 的语法。

OIL 文档(使用 OIL 表达的本体)包括容器(Container)和定义部分。容器部分类似于 XOL 的模块部分,而定义部分担当 XOL 的类和槽部分。OIL 没有实体部分。

容器部分的目的在于描述诸如名字(Name),作者(Author),主题(Subject)和描述(Description)等本体的属性和关于本体的元数据。元数据采用都柏林核心元数据元素集(Dublin Core Metadata Element Set)定义的 RDF 词汇。下面是一个容器的实例:

```
<ontology-container>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:dc="http://purl.oclc.org/dc#"
        <rdf:Description about = "">
            <dc:Title>Media Ontology</dc:Title>
            <dc:Creator>unknown</dc:Creator>
            <dc:Subject>books, videos, dvd</dc:Subject>
            <dc:Description>Ontology about media </dc:Description>
            <dc:Type>ontology</dc:Type>
            <dc:Format>text/xml</dc:Format>
            <dc:Indentifier>http://pillango.sirma.bg/sample-ontology.xml</dc:Indentifier>
            <dc:Language>OIL</dc:Language>
        </rdf:Description>
    </rdf:RDF>
</ontology-container>
```

容器部分包含以下的元素：标题(Title)，创建者(Creator)，主题(Subject)，描述(Description)，出版者(Publisher)，研究者(Contributor)，创建日期(Date)，类型(Type)，格式(Format)，标识符(Identifier)，知识来源(Source)，语言(Language)，与其他本体的关系(Relation)和版权(Rights)。

与 XOL 不同，在 OIL 中不能够定义实体，OIL 缺少任何初始数据类型的概念，例如：没有 number，string 和 integer 的概念。XOL 只支持集合类型，而 OKBC 定义了集合(Set)，列表(List)和无序的单位组(Bag)。OIL 中的槽定义与 XOL 有所不同。例如，不支持 numeric-minimum 和 numeric-maximum 等面。

可以指定 OIL 中的槽具有对称(Symmetric)或传递(Transitive)等特性。例如，我们能够定义槽 is-ancestor 具有传递性，定义 has-spouse 具有对称性。此外，在 OIL 中可以使用 subslot-of 槽将槽组织成为层次结构。每个槽有一个惟一的名称(Name)，可选的文档(Documentation)，域(Domain，指其实体能够赋值给槽的类)，范围(Nange，其实体可以是槽值的类)和属性(传递性或对称性)。我们能够使用 OIL 重写 XOL 实体的槽定义：

```
<ontology-definitions>
    <slot-def>
        <slot name="AUTHOR"/>
        <documentation>The name of the book/video author </documentation>
        <domain>
            <OR>
                <class name="BOOK" />
                <class name="VIDEO" />
            </OR>
        </domain>
    </slot-def>
    <slot-def>
        <slot name="ISBN"/>
        <documentation>The name of the book author </documentation>
        <domain>
            <class name="BOOK" />
        </domain>
    </slot-def>
    <slot-def>
        <slot name=" PUBLISHER "/>
        <documentation> The publisher of the media </documentation>
        <domain>
            <class name="BOOK" />
            <class name="VIDEO" />
            <class name="DVD" />
        </domain>
    </slot-def>
    ….
    other slot definitions
```

```
    ….
    ….
    class definitions
    ….
</ontology-definitions>
```

值得指出的是:域的槽值是一个类表达式(Class Expression)——使用 AND, OR 和 NOT 等布尔操作符连接的类。这一特性来自于 OIL 的描述逻辑基础,XOL 和 OKBC 不支持这一特性。另一个有趣的现象是,在 OIL 中不能够在槽定义(类似于槽)中定义集的势/值(Cardinality/Value)和类型/其他(Type/Etc),而是定义在使用槽的类的槽约束(类似于面)中。

类定义由类名字、文档、父类名字集合和槽约束组成。与 XOL 不同,OIL 中的类可能是初始的(Primitive)、定义的(Non-primitive)的或默认为初始的。如果类是初始的,那么它的定义(槽约束和父类成分)对于类的实体是一种必要非充分的条件。相反,如果类是定义的,那么类的定义对于类的实体是一种充分必要条件。

上节 XOL 本体实例的 OIL 表示为:

```
<ontology-definitions>
    ….
    slot definitions
    ….
    <class-def>
        <class name="STRING" / >
        <documentation>strings</documentation>
    </class-def >
    <class-def>
        <class name="MEDIA" / >
        <documentation>The class of all media types</documentation>
    </class-def >
    <class-def>
        <class name="PERSON" / >
        <documentation>The class of all persons (used for the AUTHOR slot)
        </documentation>
    </class-def >
    <class-def>
        <class name="MAN" / >
        <documentation>The class of all male humans</documentation>
        <subclass-of>
            <class name="PERSON" />
        </subclass-of>
    </class-def >
    <class-def>
        <class name="WOMAN" / >
        <documentation> The class of all female humans </documentation>
        <subclass-of>
```

```
        <AND>
            <class name="PERSON">
            <NOT>
                <class name="MAN" />
            </NOT>
        </AND>
    </subclass-of>
</class-def >
<class-def >
    <class name="ORGANISATION" />
    <documentation>The class of all orgs</documentation>
</class-def >
<class-def >
    <class name="COMPANY" />
    <documentation>The class of all companies</documentation>
    <subclass-of>
        <class name="ORGANISATION" />
    </subclass-of>
</class-def >
<class-def >
    <class name="BOOK" />
    <documentation>The class of all books</documentation>
    <subclass-of>
        <class name="MEDIA" />
    </subclass-of>
    <slot-constraint>
        <slot name="AUTHOR"/>
        <value-type>
            <class name="PERSON" />
        </value-type>
        <max-cardinality>
            <number>3<number>
            <class name="PERSON">
        </max-cardinality>
        <min-cardinality>
            <number>1<number>
            <class name="PERSON">
        </min-cardinality>
    </slot-constraint>
    <slot-constraint>
        <slot name="ISBN"/>
        <value-type>
            <class name="STRING" />
        </value-type>
```

```
            ...
            other facets
            ...
        </slot-constraint>
        <slot-constraint>
            <slot name="PUBLISHER"/>
            <value-type>
                <class name="COMPANY" />
            </value-type>
            ...
            other facets
            ...
        </slot-constraint>
    </class-def >
    ...
    other class definitions (VIDEO, DVD)
    ...
<ontology-definitions>
```

需要注意的问题：

由于在 OIL 中没有初始的数据类型，我们定义了 STRING 类用于 ISBN 槽的值。

我们添加了 PERSON 类，并且定义了它的两个不相交的子类——MAN 和 WOMAN。在 XOL 中不能够表示不相交的类。

OIL 允许将类定义在与面相关的势中，增强了 OIL 的表示能力。例如：我们可以定义某些槽对于类 X 具有势 N，同时对于类 Y 具有势 M。当然，这有可能导致不一致性。

XOL 与 OIL 的对应关系如表 3.2。

表 3.2　XOL 与 OIL 的对应关系

XOL 概念	OIL 概念
Module/ontology/kb/database/dataset 元素	Ontology-container 元素
X	Relation(在 Ontology-container 内引用其他本体)
Name(Module/...内的元素)	Ontology-container 内的 dc:title 元素
X	Import 元素(被包含的本体)
Class 元素	Class-def 元素
Name(class 或 slot 内的元素)	Class 或 slot 元素的 name 属性
Documentation(class 或 slot 中的元素)	class-of 或 slot-of 内的 Documentation
X	Class-def 的 type 属性(初始的/定义的)
Subclass-of 元素	Subclass-of 元素
X	类表达式
X	不相交的类
Value(slot-values 内的元素)	solt-constraints 内的 Has-value
初始的数据类型	X
Slot 元素	Slot-def 元素
Slot 内的 domain 元素	Slot-def 内的 domain 元素
Slot-value-type 槽	Slot-def 内的 Range 或 slot-constraints 内的 value-type

(续表)

XOL 概念	OIL 概念
Slot-inverse 槽	Slot-def 内的 inverse 元素
X	Slot-def 内的 properties 元素
X	Slot-def 内的 subslot-def
Individual 元素	X
Instance-of 元素	X
Template 类型槽内的 Max-cardinality	slot-constraints 内的 max-cardinality
Own 类型槽内的 slot-max-cardinality	X
Numeric-mininum/slot-numeric-mininum	X
Slot-collecton-type/collection-type	X

OIL 的一个重要特征是:使用 OIL 定义的本体(槽和类定义)能够被映射到 SHIQ 描述逻辑的公理,因此能够借助 FaCT(Fast Classification of Terminologies)系统执行可靠性和完整性推理。

3.4.2.3 RDFS(RDF Schema)作为本体语言

在本节中介绍 OIL 的作者对 OIL 和 RDFS 关系进行的分析。RDFS 通过添加本体语言中常用的建模元语对 RDF 进行进一步地扩展,这些元语包括类、类继承、属性继承、域和范围。域和范围用于定义实体属于哪些类。

在 RDFS 中 superclass-of 层次定义如图 3.15。

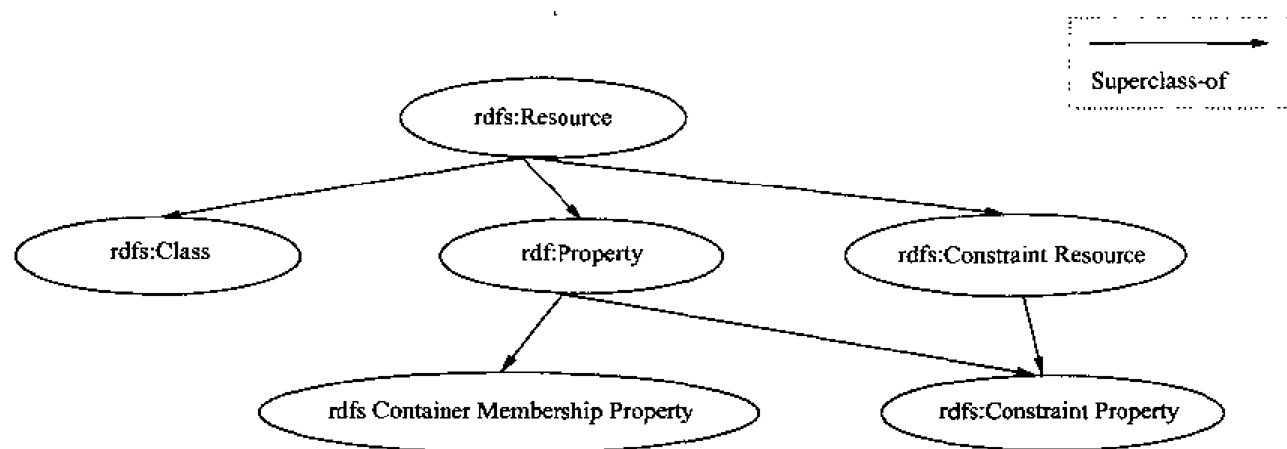


图 3.15 superclass-of 的层次定义

instance-of 关系定义如图 3.16。

请注意在"rdf"和"rdfs"的名字空间并没有清晰地分离。例如:RDFS 并不仅是 RDF 的扩展,因为 REF(ref:Property)也依赖 RDFS。同时,RDFS 除文字方面以外没有定义初始数据类型。

Rdfs:Class 用于定义类或概念,它们类似于 XOL 中的 class 元素和 OIL 中的 class-def 元素。类通过 rdfs:subClassOf 属性组织为层次结构。RDFS 没有 primitive/defined 类的概念,不能够直接表达类似于 OIL 的复杂类表达式。资源类似于实体,通过 rdf:type 属性表示属于相同类的实体,它类似于 XOL 中的 instance-of 元素。

每个类都有与之相关的属性和约束集合,约束也是一种属性。属性由 rdf:Property 表示,类似于 XOL 中的 slot 和 OIL 中的 slot-def。属性通过使用 rdfs:subPropertyOf 组织为层次结

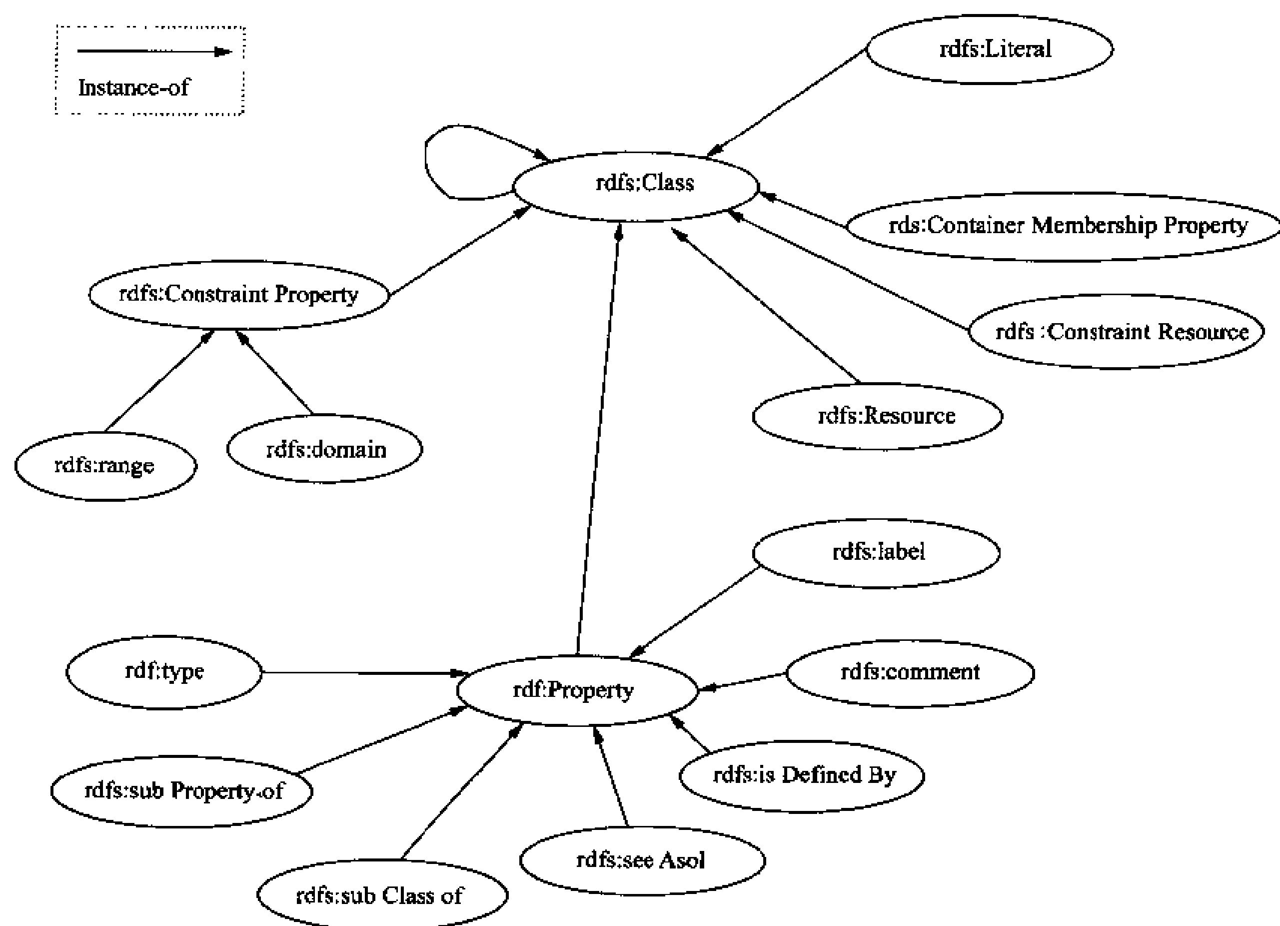


图 3.16　instance-of 关系定义

构，类似于 OIL 中的 subslot-of。RDFS 中的属性类似于 XOL/OIL 中的槽，能够独立于类定义。

Rdfs:subClassOf 和 rdfs:subPropertyOf 的不足之处是不能够在 class/slot 层次图中形成循环，因此不能够表示 classes/slots 之间的相等关系。

Rdfs:range 和 rdfs:domain 的标准约束用于定义属性的合法取值以及能够具有该属性的类。Rdfs:range 约束最多只能够在类定义中出现一次。由于没有复杂类表达式，可以通过创建一个人工的父类为属性定义合法的类取值。

下面是使用 RDFS 定义的一个简单的类以及它的属性：

```
<rdf:Property rdf:ID="CREATOR">
    <rdfs:comment>the agent who created the entity</rdfs:comment>
    <rdfs:range rdf:resource="#PERSON" />
    <! -domain is unspecified -->
</rdf:Property>
<rdf:Property rdf:ID="AUTHOR">
    <rdfs:comment>The Author of the book/video </rdfs:comment>
    <rdfs:subPropertyOf rdf:resource="#CREATOR" />
    <rdfs:domain rdf:resource="#BOOK"/>
    <rdfs:domain rdf:resource="#VIDEO"/>
</rdf:Property>
```

```
<rdf:Property rdf:ID="PUBLISHER">
    <rdfs:comment>The publisher of the book/video/dvd</rdfs:comment>
    <rdfs:domain rdf:resource="#MEDIA"/>
    <rdfs:range rdf:resource="#ORGANIZATION" />
</rdf:Property>
<rdf:Property rdf:ID="ISBN">
    <rdfs:comment>The ISBN of the book </rdfs:comment>
    <rdfs:domain rdf:resource="#BOOK"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" / >
</rdf:Property>
...
```

上面的定义类似于 XOL 中的槽部分，类似于 OIL 中包含槽定义的本体定义部分的一部分。下面是类定义的实例：

```
<rdfs:Class rdf:ID="MEDIA">
    <rdfs:comment> The class of all media types </rdfs:comment>
</rdfs:Class>
<rdfs:Class rdf:ID="PERSON">
    <rdfs:comment> The class of all persons </rdfs:comment>
</rdfs:Class>
<rdfs:Class rdf:ID="ORGANIZATION">
    <rdfs:comment> The class of all organizations </rdfs:comment>
</rdfs:Class>
<rdfs:Class rdf:ID="BOOK">
    <rdfs:comment> The class of all books </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#MEDIA" />
</rdfs:Class>
<rdfs:Class rdf:ID="VIDEO">
    <rdfs:comment> The class of all videos </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#MEDIA" />
</rdfs:Class>
<rdfs:Class rdf:ID="DVD">
    <rdfs:comment> The class of all DVDs </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#MEDIA" />
</rdfs:Class>
…. Definitions for other classes like PERSON, ORGANIZATION ,...
```

上面的定义类似于 XOL 中的类部分，类似于 OIL 中包含类定义的本体定义部分的一部分。下面是某些实体定义的实例：

```
<rdf:Description rdf:ID="Knopf">
    <rdf:comment> Knopf publishing company </rdf:comment>
    <rdf:type rdf:ID="#ORGANIZATION" />
</rdf:Description>
<rdf:Description rdf:ID="Wharton">
```

```
        <rdf:comment> William Wharton </rdf:comment>
        <rdf:type rdf:ID="#PERSON" />
        <FIRST_NAME> William </FIRST_NAME>
        <LAST_NAME> Wharton </LAST_NAME>
    </rdf:Description>
    <rdf:Description rdf:ID="Birdy">
        <rdf:comment> cool book </rdf:comment>
        <rdf:type rdf:ID="#BOOK" />
        <AUTHOR rdf:resource="#Wharton" />
        <PUBLISHER rdf:resource="#Knopf" />
        <ISBN>0679734120</ISBN>
    </rdf:Description>
```

注意 RDFS 没有定义关于表示本体的文档结构。例如：RDFS 没有像 XOL/OIL 中的分离的 module/ontology-container 和 class/ontology-definition 部分。都柏林核心元素集可以用于形成本体的描述。XOL，OIL 和 RDFS 的对应关系如表 3.3。

表 3.3 XOL，OIL 和 RDFS 的对应关系表

XOL 概念	OIL 概念	RDF 概念
Module/ontology/kb/database/dataset 元素	Ontology-container 元素	X
X	Relation（在 Ontology-container 内引用其他本体）	X
Name（Module/...内的元素）	Ontology-container 内的 dc:title 元素	X
X	Import 元素（被包含的本体）	X
Class 元素	Class-def 元素	rdfs:Class 元素
Name（class 或 slot 内的元素）	Class 或 slot 元素的 name 属性	rdfs:Class 内的 rdfs:ID 属性
Documentation（class 或 slot 中的元素）	class-of 或 slot-of 内的 Documentation	rdfs:Class 内的 rdfs:Comment
X	Class-def 的 type 属性（初始的/定义的）	X
Subclass-of 元素	Subclass-of 元素	Rdfs:subClassOf 元素
X	类表达式	X
X	不相交的类	X
初始的数据类型	X	X
Slot 元素	Slot-def 元素	rdfs:Property 元素
Slot 内的 domain 元素	Slot-def 内的 domain 元素	rdfs:Property 内的 rdfs:domain 元素
Slot-value-type 槽	Slot-def 内的 Range 或 slot-constraints 内的 value-type	rdfs:Property 内的 rdfs:range 元素
Slot-inverse 槽	Slot-def 内的 inverse 元素	X
X	Slot-def 内的 properties 元素	X
X	Slot-def 内的 subslot-def	rdfs:Property 内的 rdfs:subPropertyOf
Individual 元素	X	rdfs:Description/rdfs:Resource 元素
Instance-of 元素	X	描述内的 rdfs:type 元素
Value（slot-values 内的元素）	slot-constraints 内的 Has-value	表示槽的元素
Template 类型槽内的 Max-cardinality	slot-constraints 内的 max-cardinality	X
Own 类型槽内的 slot-max-cardinality	X	X
Numeric-mininum/slot-numeric-minunum	X	X
Slot-collection-type/collection-type	X	?

3.4.2.4 本体到DTD的映射算法

本体到DTD的映射主要是采用XML语法描述本体中概念集合和概念之间的关系集合。DTD为文档结构定制了一套规则，规定了XML文档中的元素类型、属性、文档中的实体及相互关系等。由于XML语法难以描述和应用规则，在XML文档内容需要推理支持时，推理引擎对XML文档自描述信息中涉及的概念应用规则集合中相关的规则进行推理。本节主要介绍采用描述逻辑形式化表示的本体的映射方法，具体采用贝尔实验室开发的DL支持系统CLASSIC。由于CLASSIC支持DL的标准构造符，因此提出的映射方法也适用于其他DL系统。

定义1：形式化的本体定义为四元组，Ontologly =（meta _ info，CONCEPT，RELATION，RULE），其中meta _ info是本体的名称，创建者，创建时间，创建依据等本体的元信息；CONCEPT是组成本体的概念集合；RELATIOAN是本体中概念之间的关系集合；RULE是在本体中普遍成立的规则集合。

DL系统采用概念（Concept）、角色（Role）和规则（Rule）描述领域本体，所有概念都是某个概念的子女概念，这一初始概念可以是系统默认设定的，也可以是用户自定义的，DL系统支持继承关系的定义和推理，支持概念同时继承多个父概念。CLASSIC中概念描述语法的片段为：

```
ClassicDescription ::=  ClassicThing |  ClassicConcept |
          (andClassicDescription+) |(oneOf ClassicIndividual+) |
          (atLeast PositiveInteger Role) |  (atMost NonNegativeInteger Role)
          (fills Role ClassicIndividual+)|  (fills Role HostIndividual+) |
          (all Role Description) |    (testC ClassicTestGenerate Parameter*)
```

结合XML DTD定义的相关技术，映射方法的基本思路为：

(1) 本体中的每个概念在DTD中生成一个元素类型。

(2) 概念的每个角色在DTD中生成一个子元素。

(3) 如果角色表示的是与其他概念的关系，角色元素的内容模型为相关的概念元素，否则角色元素的内容模型为PCDATA类型。

定义2：不继承任何概念的概念为初始概念RootConcept，在特定的本体中初始概念是惟一的。

定义3：本体中概念和概念之间的关系组成本体继承关系图$G=(V_{concept}, E_{nherit_relation})$，其中$V_{concept}$为对应于本体中所有概念的结点集合，$E_{inhent_relation}$为对应于本体中继承关系的有向边集合，有向边$<V_{concept_i}, V_{concept_j}>$表示概念$concept_j$为概念$concept_i$的父结点。继承关系集合inherit _ relation ⊆ RELATION。

定义4：继承关系图G中，结点$V_{concept_i}$到$V_{rootconcept}$经过的有向边的序列为$V_{concept_i}$的一种继承关系路径，路径长度$L^k_{concept_i, rootconcept}$为这条路径上的边数，

$\max\limits_{k\in 1,\cdots,m} L^k_{concept_i, rootconcept} = L_{concept_i, rootconcept}$，m表示有m个继承关系路径。

本体到DTD映射算法的具体步骤如下，：

按照概念的继承关系，采用广度优先的方法生成本体中所有概念的集合$AllConcept = \{concept_1, \cdots\cdots, concept_n\}$，在AllConcept中，如果$i<j$，那么$L_{concept_i, rootconcept} \leqslant L_{concept_j, rootconcept}$。

生成 AllConcept 中每个概念的祖先概念的集合 Anc{$concept_i$}和每个概念的子孙概念的集合 Dec{$concept_i$}，$i=1,\cdots,n$。

在 DTD 中使用参数实体，定义体现于 Dec{$concept_i$}中的每个概念的继承关系，如果 Dec{$concept_i$} = {$concept_1,\cdots,concept_p$} 生成对应的参数实体为：

<! ENTITY % $concept_i$"$concept_i$|$concept_1$|⋯|$concept_p$">

按照概念出现的顺序，生成每个概念和角色的 DTD 定义。

RoleRestrictions 为本体中单个概念角色定义的集合：

RoleRestrictions = {$Role_1$ _ Description，⋯，$Role_q$ _ Description}

其中，$Role_j$ _ Description = <role _ name，X-Y，selection，role _ type>

X 表示角色在 $concept_i$ 中至少要出现的次数，$Y \geq X$，Y 为 * 时，表示角色在 $concept_i$ 中出现的次数没有上限；Selection 为角色可选的取值集合；Role _ type 表示角色 role 的类型。

每个概念的角色为所有祖先概念角色集合和概念定义中角色定义的并集，即如果 $concept_i$ 的祖先概念集合为 Anc{$concept_i$}，

Anc{$concept_i$} = {$concept_1,\cdots,concept_q$}，概念 $concept_i$ 定义中角色定义为 RoleRestrictions′，那么概念 $concept_i$ 的 RoleRestrictions 为：

RoleRestrictions = RoleRestrictions′ ∪ $concept_1$. RoleRestrictions ∪ ⋯ ∪ conceptq. RoleRestrictions

(1) 生成概念在 DTD 中对应的元素列表，概念的角色转换为元素的子元素：

<! ELEMENT $concept_i$(role _ $name_1$ $^{*?+}$，⋯，role _ $name_q^{*?+}$)> ，role _ $name_j$ 表示 $Role_j$ _ Description 的 role _ name，如果 X= 0，Y=1 上标为"?"，X>0，上标为" + "，X=0，Y>1 上标为" * "。

同时生成元素在 DTD 中的惟一标识属性 ID，ID 在同一 XML 文档中取值必须惟一：

<! ATTLIST $concept_i$ OID ID ＃IMPLED>。

(2) 生成每个角色对应的子元素定义：

①如果 role _ type 为标量类型(STRING，INTEGER，REAL，BOOLEAN 等)，$role_j$ 相应的元素定义为：

<! ELEMENT role _ $name_j$(＃PCDATA)>。

②如果 role _ type 为本体中的其他概念 $concept_X$，$role_j$ 相应的元素定义为：

<! ELEMENT role _ $name_j$(＃PCDATA)>。

同时生成子元素 IDREF 类型的属性，通过与 $concept_X$ 元素 ID 值匹配，限制子元素的取值类型：

<! ATTLIST role _ $name_j$ OIDREF IDREF ＃IMPLED>，

<! ATTLIST role _ $name_j$ Type (%$concept_X$;) "$concept_X$">。

由于 XML 语法限制 IDREF 类型的属性值必须为同一篇 XML 文档中另一元素的 ID，有时将 IDREF 参考的元素都放在同一篇 XML 文档中会导致 XML 文档规模过大；同时，导致同一元素要在不同的 XML 文档中重复定义。因此，我们应用 Xlink 技术，定义虚元素解决 ID/IDREF 文档间的匹配问题。虚元素定义格式如下：

<! ELEMENT Virtual _ $concept_X$(＃PCDATA| % $concept_X$;)>

<! ALLLIST Virtual _ $concept_X$ OID ID ＃IMPLIED>

<! ALLLIST Virtual _ $concept_X$

```
xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink"
xlink:type CDATA #FIXED "simple"
xlink:show (new | replace | embed) "new"
xlink:href CDATA #REQUIRED
>。
```

Virtual _ concept$_X$ 实例的 ID 值等于 IDREF 要参考的其他 XML 文档中 concept$_X$ 类型元素实例的 ID 值。

第 4 章　语义 Web 的结构和思想

语义 Web 将是新一代 Web 的发展目标之一，了解语义 Web 的结构和思想是应用语义信息模型及研究语义 Web 的基础。本章主要介绍语义 Web 的体系结构、基本概念和主要思想以及主要的设计原则。

4.1　XML 下的 Web 的体系结构

Web 是由简单的协议控制的松散的、开放的资源集合，只有 XML 能出色地实现这一目标，它能够构建真正的、由开放标准和自描述数据控制的多层分布式系统。

传统的 Web 的体系结构(如图 4.1 所示)是客户端程序的一个浏览器，它充当着浏览者的代理角色。浏览器将对页面的请求发送给 HTTP 服务器，这个请求会跟随着一系列的参数名称和值。服务器会通过 CGI 脚本或 ASP 代码来动态生成 HTML 以满足这类请求。

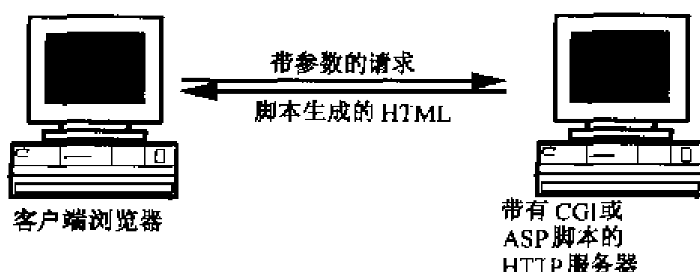


图 4.1　传统 Web 的结构

当然，这种结构足以正常地工作，但它确实有一些重要的限制。如果客户端浏览器或者程序将一个 XML 文档作为请求发送给服务器(如图 4.2 所示)。那么这些对我们来说有什么益处呢？首先，我们不再局限于基于浏览器的客户端。XML 本身就是数据，而且可以由程序任意地控制。同样的数据，既可以设定其样式化以便在浏览器中显示，也可以交给一个代理进行后台处理。在这个机制中，XML 文档无需假设数据的最终用途。

在这种机制下，服务器端的应用程序与客户端的耦合程度要松散得多，因为程序具备了找出 XML 文档的结构的能力。这样一来，富有创新意义的应用程序就可以根据程序的要求编写结构新颖的各类文档，应用程序也无需为每一种新的文档类型编制定制的软件。

在不久的将来，网络中的服务器，客户机和应用程序所进行的处理都将使用这种机制交换数据。事实上，任何一种平台都支持这种机制，它使用简单，能够处理来自不同数据源的标记数据。应用程序的开发者可以使用来自非传统数据源或其他服务器的数据以满足客户端的请求。自此，Web 开发已经从客户机/服务器计算体系迈向真正的多层模式。

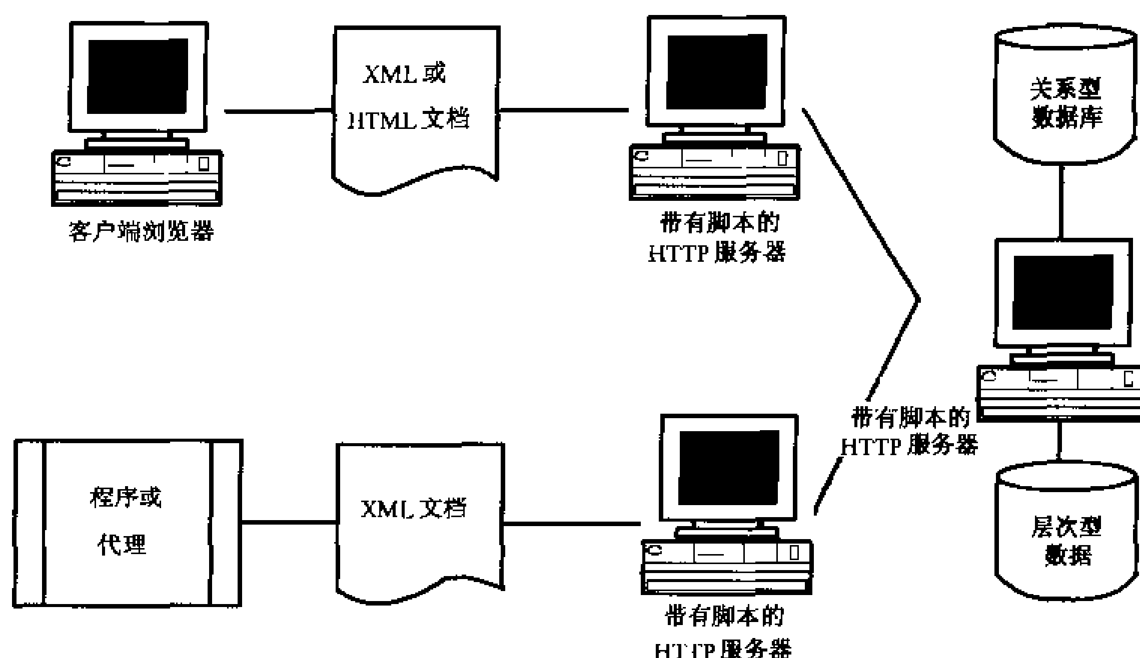


图 4.2 XML 下的 Web 的结构

4.2 Web 信息空间

1. 目标

W3C 联盟明确规定的任务是将 Web 引向其"完全潜力"。Web 可以定义为一个可网络化访问的信息世界,并且将该"完全潜力"分为两部分,首先将其作为一种人与人交流的手段,其次是软件主体可在其中工作并成为人类的工具的一个空间。

在这节中,将首先阐明这一空间的性质,然后看一看它作为人类通信的工具如何成为机器推理的工具。

2. 基础:全球 Web

Web 体系结构最基本的定义是全球资源标识(Universal Resource Identifier,URI)。Web 上的任何东西都要惟一地由一个不透明的字符串(一个 URI 和一个分段标识符)来表示,这一原则是普遍性的核心。

3. URI 方案

新的 Web 发展指导原则是 Web 的发展应遵循 URI 的通用定义和语法,没有充分理由则不引入新的 URI 方案,并且不引人任何使其成为 URI 超集的不同方案。

最小(Minimalist)设计原则要求 URI 超空间本身对任何特定的 URI 方案在诸如同一性、持续性和可解析性等性质方面加以最小约束。

4.2.1 HTTP 空间

最著名的 URI 空间是 HTTP 空间,它具有灵活而一致的概念和丰富的资源信息。

HTTP 空间包括两部分,一部分是使用域名系统(Domain Name System,DNS)的层次化表示,另一部分是不透明字符串,由拥有域名的本地特权定义其意义。

HTTP 空间的 Achilles 根是仅有的集中部分,负责 DNS 树的根的所有权和管理。对整个

HTTP Web 来说，DNS 根是一个关键性的资源，以一种公平的方式将其作为一个整体进行管理是十分重要的。

其他方面的研究工作包含了种种与 HTTP 类似或不类似的许多“命名”问题，正在使用的“URN”一词就是用于这样的主题的。如果创建一个新的空间，它就可用做通用 URI 空间的子空间，从而使得 Web 的通用性得到保持，也使得新空间的功能可用于所有资源，这一点是极为重要的。

人们可以期待 HTTP 变得更加成熟，以提供更加标准的解析 HTTP 地址的方法，同时保持相同的(层次加不透明字符串)地址空间。

目前在 Internet 上，HTTP 用于 Web 页面，SMTP 用于 E-mail 消息，NNTP 用于网络新闻。然而所传输的对象基本上都是 MIME 对象，且协议的选择经常是由用户错误地做出的。“系统”(机器、网络和软件)自适应地决定使用哪一类协议来高效地发布信息是一种理想的情况。

4.2.2 内容和远程操作

URI 规范有效地定义了一个空间，它是标识符(URI)和资源之间的一个映射。这完全是因为理论上需要定义这样一个空间。为了使该空间的内容可用，解析标识符的运算就成为一个基本的需求。在 HTTP 中这是一个“GET”操作。在 Web 体系结构中，GET 因此而具有了一个特殊的地位。

除 GET 以外其他任何幂等方法的引入都是不正确的，因为这样的一个操作会形成一个独立的地址空间，破坏了统一性。

HTTP 的扩展，即包含一个自适应系统，作为一个实际的或预期的需求的功能，用于信息的预发布和用于副本的定位，使目前混乱的推拉协议(SMTP、NNTP、HTTP 以及增加了“通道”技术的 HTTP)自然实现优化。这是一个结果不寻常且非常适合研究的领域。

HTTP 实际上将一个基本的传输协议结合了用于一部分种类的“元数据”——即关于信息的格式。这是来自 SMTP 的一个历史性的继承，并且是应由一个更清晰的基本 HTTP 功能和更丰富的元数据世界之间的区别来代替的体系结构特征。

1. 远程操作

HTTP 开始时是被设计成针对一个对象进行远程操作并具有一个灵活的方法集的协议。分布式面向对象系统，如 CORBA、DCOM 和 RMI，具有独特的功能并与 Web 地址空间相区别的情况引起了某种问题，HTTP-NG 组织对未来 NG 发展进行了多方面调查，包括远程过程调用(RPC)和已有 Web 协议的可能的统一。这一研究结束了，但还没有产生进一步工作的推动力，而 XML 在新协议上的使用可能超越这项工作。

HTTP 和 XML 都遇到了可扩展性的问题。XML/RDF 扩展模型足以满足 RPC 的需要，并且 RPC 消息是结构化文档的特例。采用整个 RPC 系统并将它在 RDF 模型中表示出来将会是非常合理的。当然，用于 XML 的二进制格式(即便是压缩的)将是高效传输所要求的。但 RDF 的可发展性才是 RPC 所需要的。

2. 消息和文档

新协议应从消息交换的意义来定义，这里的消息就是 XML 文档，实际上也就是 RDF 文档。

商务协议是使用放在 Web 上的文档或是用因特网消息通过 SMTP 或 HTTP 发送的文档

来定义的。可以认为消息和文档是同样的，不仅能在给文档签名的地方提供消息签名，还能在给消息二进制编码的地方提供文档二进制编码。

那么，对于 HTTP 发展的理想目标应包括：

(1) 一个允许多个并发消息交换的协议。

(2) 一个数据类型标准，用于与具有名称空间的 XML 文档一样普通和一样可扩展的对象。

(3) 一个模式系统，允许任何(CORBA、DCOM、RMI 等)RPC 接口被定义，并具有用于 RPC 传输的隐含的标准高效的格式。

(4) 对 RPC 状态转换协议的扩展，允许 Web 应用所需的异步性(双向性、流及异步调用……)。

(5) 复杂的社会传媒(Wcb)协议在新的 RPC 功能上的实现。

3. 访问协议的扩展

W3C 在协议扩展方案上的工作已经开展了很长时间，但没有在 HTTP 1.1 中大规模采用，目前采取补充规范说明的形式。许多特征如 PICS 或 RTSP 若在早期 HTTP 版本中定义，本可以从中受益。

下一代协议如 HTTP-NG 的扩展明显是一个重要的问题，但从数据格式的可扩展性中得到的经验将有希望提供足够强大的工具来直接加速并及时地被 HTTP-NG 组所使用。

协议或数据格式的规范说明必须考虑并区分必需的和可选的扩展。能在 XML 中做到这一点的通用工具是非常需要的。

4.2.3 数据格式

1. 格式协商

当 URI 体系结构被定义，并且人们已经使用了至少一种可解析的协议后，人们对一个可互操作的全球超文本系统的需求至少是可用于资源内容或 Web 对象的一个公共格式。

Web 的初始设计假定了将会继续有大量的私有数据格式的增长，因此 HTTP 被设计为具有在客户端和服务器端协商公共格式的特征。

Web 目前完全都是用户知道的数据格式，数据格式的显式的用户选择，使其更加复杂化并隐藏了信息的本质。

2. MIME 类型

在 HTTP 中，数据的格式是通过“MIME 类型”定义的。形式上这涉及由 IANA 维护的中心注册表，但在体系结构上，这是一个不必要的中心控制点，Web 本身应该能作为一个新类型库使用。

目前 Web 体系结构需要 URI 分段标识符的语法和语义成为 MIME 类型的一个功能。

3. 结构化文档的通用语法：XML

尽管采用非结构化文档共享一个通用语法是个好办法，但还是需要某种更为简单的东西。XML 就是答案。

尽管任何人都可以在一种新语言中自由使用任何语法，但共享语法有非常显著的优点，以至新语言都用 XML 编写。除了共享工具、解析器和理解力所产生的高效率之外，它还承担了以国际化的方式提交给 XML 的工作，以及可扩展性。

4. 名称空间

在一个分布的开发者团体中进行的对技术发展需求的调查显示 XML 语言必须具有下列特征：

(1) 必须能够精确地将语言(符号集、语法及语义)定义为第一类对象。

(2) 必须能以多种语言混合形成文档。

(3) 每个文档应通过它所用语言的 URI 成为自定义的。

(4) 必须能够理解(至少是部分理解)一个文档子集。

这些需求依赖于以人为目标的数据格式或是机器不可理解的(语义)数据格式的发展。

当定义一种新的语言后，通常应使用 XML。当使用它时，新语言或扩展已有语言的新特征必须作为一个新的名称空间被定义。一个名称空间的 URI 必须用于识别语言。XML 应被认为是包含 XML1.0 和名称空间两者。

1999 年提供了 XML 和 RDF 模式语言可用的草案形式。新的名称空间必须设计成为模式语言草案的测试案例，并且不依赖于 DTD 的功能。

语言规范应定义该语言能以何种方式扩展。这一般包括定义其子类型可在以后的语言中创建的元素的类型。原始语言的结构化约束将定义新语言如何可依照语法与旧语言融合，旧规范的语义将定义新元素应如何在规范的语义层解释。将来这一领域的工作需要明确这一点以及它是如何在模式语言中表达的。

概括起来，新的语言(名称空间)可以通过两种方式引入。首先，作为一个全新的应用，其次，作为已有的应用，如 HTML 或 RDF 的扩展。在后一种情况下，诸如用于可读文档的样式表和用于逻辑文档的推理规则，这样的语言将在给定语义层定义新语言的解释。

名称空间文档是语言发布者保存关于名称空间的定义资料的地方。模式语言对于这一点是非常理想的。在 Web 上有一个自描述文档可获得大量价值。自描述文档的第一级是在 XML 模式(结构)层上做的，后面各级是给出语义信息的。

语言像资源一样可以是活的也可以是冻结的。要使语言成为活语言是危险的，要求有类似 HTML 的发散度。即便语言是冻结的，名称空间文档也可以在新语言可用于表达不同形式的语义时改变。例如，名称空间文档可包括或链接到：

(1) 依句法的约束(如在 XML 模式中)。

(2) 属性的范围和域(如在 RDF 模式中)。

(3) 用于语言表现的默认的或强制性的样式表(如在 CSS 或 XSL 中)。

名称空间文档显然可以有多种语言的混合。

4.2.4 人类可读信息

人们所说的人类可读信息指的是传统意义上用于人类消费的文档。尽管这些文档可以由机器传输、提交、分析和索引，但理解它们思想的是人工智能问题，不将它作为 Web 体系结构的一部分来提及。当谈到机器可理解的文档时，人们指的是显式地为机器推理准备的数据：语义 Web 的一部分。

1. 形式与内容的分离

SGML 组所提倡的一个体系结构规则是形式与内容的分离。它是 Web 体系结构的一个实质部分，使上面提及的设备独立性成为可能，也有助于处理和分析。HTML 中表现信息的加入使其可使用样式表，但违背了这一规则。该规则适用于除 HTML 以外的许多规范说明：

在数学标记语言(Math Markup Language,MathML)中存在两个层次的语言,一个与数学上的意义有关,另一个仅标明物理外观。

2.图形

用于人类可读文档的不同语言的发展是可以相对独立的。所以2D图形语言(如PNG和SVG)是真正独立于3D语言(如VRML)及文本语言(如HTML和MathML)的。当考虑到样式、字体、颜色和国际化这些方面时就需要协调,应该有单一的用于所有语言的通用模型。

PNG作为一种压缩编码引入,在GIF上既从技术上(颜色、灵活度和透明度)也从政策上(没有阻碍)进行了改善。要求SVG作为一种通用格式,需要一种面向对象的绘图XML语言的建议。

3.HTML

通用文档语言的价值相当大,以至HTML获得了Web上的主导地位,但它没有扮演一个基础的关键角色。Web应用程序需要能够处理HTML,因为它是Web的连接组织,但它在体系结构上并没有什么特别的位置。

HTML已经从自由扩展的“忽略你不理解的东西”规则中受益也受害了。未来的计划是将HTML从一个SGML应用移植到作为一个XML名称空间来定义。第一步是对具有HTML 4.0特征但实际上是XML文档的规格说明。

4.XHTML转换

从今天的HTML到未来基于XML的HTML的转换策略比较难。它受到许多约束:

(1) 目前许多网页做得较差,不符合HTML 4.0标准,同样也不符合SGML。

(2) 浏览器必须保持在很长时期里能够读这些已有网页。

(3) 许多现有浏览器不能解析XML。

(4) 有一种方式即“XHTML”,能够编写形式良好的XML文档,浏览器上显示为HTML并能被正确地解析。

(5) 人们可以通过名称空间声明区分出一个旧的HTML文档和一个XML文档。

转换策略准备开始在一个站点内部使用XML,并且用于内部文档,同时将网页格式化为XHTML。这将使网站能够使用许多XML工具,刺激了XML工具的市场。

第二阶段是网站将HTML页转换为XHTML页。这样做的一个动机是能够直接在站点上使用XML工具(编写XHTML页将需要一个专门的转换器)。这将产生一个形式良好的XML页的基础,并有利于在浏览器和搜索引擎中加入XML解析。在转换阶段,任何找到XML文档的XML浏览器都必须假定它是带有名称空间的形式良好的XML。任何不是XML的东西都可能由浏览器供给已有的HTML引擎。XHTML页将是可扩展的,已有浏览器将忽视它们所不能识别的任何标记。这种转换将有希望因开放源代码的可用性而变得更容易,开放源码将采用一个典型的旧HTML页转换(大多数情况下是零损失)为一个完全合法的XHTML页。这将会使所有新工具仅接受XML,因而也就准备作为XML传播之用。

最终,需要使用其他语言或其他XML特征如Unicode的站点将有希望引起普遍的升级,直到大多数Web客户都能够处理具有名称空间的XML,并且各站点将能够坚持这点。

5.超文本链接拓扑

允许Web伸缩的一个基本折中方案是,在体系结构策略上,链接应基本上是单向的。链接初始有三个参数:源(在源文档中是隐式的)、目的和类型。类型参数能够增加语义,还没有较多地被使用,但XLINK小组有一个目标,即为其增加丰富性,尤其对大的Web页集合。然

而要注意的是，下面介绍的资源描述框架(Resource Description Framework)是一个模型，因此链接关系就如未来 Web 体系结构中的其他任何关系一样，必须是在 RDF 中可表达的。以这种方式，HTML 中和未来 XML 超文本语言中的链接关系应成为第一类对象。

XLINK 还将定义更为复杂的链接拓扑，并提出与其相关的人机接口问题，仍旧使用相同的 URI 空间并将 RDF 作为关系规范的定义语言。

6. 样式表

模块化设计原则意味着当分离形式和内容时，语言的选择应尽可能独立。HTML 已经统治了文本标记语言，但 XML 的引入打开了使用新的 XML 标记语言的大门。

样式实质上是在文档的抽象内容和它向其接受者所展示、述说、表现的物理形式之间的一个映射。

对于图形样式，级联样式表(Cascading Style Sheets, CSS)提供了一种声明文档元素表现形式的方式。它具有说明性强和可逆的优点：人们可以制作一个用于编辑应用了样式表的文档的编辑器。相反 XSLT 是一种可以生成从输入到输出结构的任意复杂的映射的转换语言。它促进了更为强大的处理，但不是普遍可逆的。对于图形表现，XSLT 可被用于映射到一个格式对象集(Formating Objects, XSL-FO)，其格式属性将成为 CSS 属性集的超集。

历史造成了 CSS 不是一种 XML 语言的事实，这是因为它出现在 XML 之前。CSS 格式属性的名称空间允许 CSS 被直观地加入任意 XML 文档，这是一个自然的发展方向，但可能因 XSL-FO 而成为不必要的。

7. 协作

使 Web 成为人们共同工作的创造性空间这一最初想法现在看来进展很慢。

这一领域非常宽广，可以被分为以下各个范围：

(1) 异步协作工具：包括论坛、工作流自动控制、注解 系统、签注和协作过滤。

(2) 实时视音频协作和 Web 的集成：包括视频寻址、HTML 中视频的集成(SMIL 等)和联合表现系统。

(3) 群编辑器(同步超文本编辑器、白板等)。

(4) 异步分布式编辑(Amaya, Jigsaw, Jigedit, WebDAV)。

建立一个保证成员间相互信任而具有充分保密性的环境是许多协作工作的前提。

上面各范围中多数是研究领域，有些领域已有了产品。W3C 成员还没有提出该领域的公共规范说明。W3C 试图使用任何基于 Web 的可用协作技术，包括 Web 上文档的分布式编辑以及自动变更跟踪。体系结构中的这一部分像留待日后扩展。

4.2.5 机器可理解信息

语义 Web(Semantic Web)是一个数据网，某种程度上像一个全球的数据库。

当看到对全球的语义 Web 的形式化表示时，最小化设计原则要求它基于一个具有普遍性的公共模型。只有当公共模型是通用的，才会使任何具有前景的应用映射到该模型上。这个通用模型就是资源描述框架(Resource Description Framework)。

RDF 的体系结构和建立在其上的语义 Web 是一个规划，还没成为现实。

(1) XML 提供了结构化文档的一个基本格式，不带特殊的语义。

(2) 基本断言模型提供了断言(属性)和引用的概念(由 RDF 模型和语法规范提供)。只要给出断言命题逻辑的语义，就允许为数据生成类实体－关系模型。

(3) 模式语言提供了数据分类，允许限制文档结构以及允许可预测的可计算处理。参见 XML 方案的结构和数据类型草案。

(4) 转换语言允许推理规则的表示，它使得一种模式中的信息能够从另一个文档中推断出来。

(5) 逻辑层使用推理和函数，将一种限定的声明语言变为一种完全图灵式的逻辑语言。这强大得足以定义所有其他的语言，并允许任意两个 RDF 引用连接在一起。我们可以将该语言看做是将所有数据系统统一起来的通用语言，就如同 HTML 是一种将所有人类文档系统统一起来的语言一样。

(6) 证明语言是 RDF 的一种形式，它允许发送一个代理到另一个断言，它允许如存取控制这样的应用程序使用通用验证引擎作为内核。

(7) 演化规则语言允许给出这样的推理规则，它允许具有某种算法的机器将文档从一种 RDF 模式转换成另一种 RDF 模式。这是技术发展的一个基本要素。

(8) 查询语言支持不同形式的查询引擎，同时标准化某些查询引擎和一种定义查询引擎的语言。

一旦有了证明语言，则数字签名的引入就将 Web 变为了信任的 Web。数字签名功能在 RDF 世界的发展原则上可以与上面各阶段并行发生。

单一数字签名格式对于 XML 文档是很重要的。RDF 逻辑层将允许已有的证书方案转换为 RDF，以及信任路径由通用 RDF 引擎来证实。

4.2.6 元数据应用

语义 Web 的第一层驱动是关于信息的信息，通常称为元数据。下面各领域是 W3C 正在或将要开发的使用 RDF 的技术实例。

(1) 同步多媒体(SMIL)。

(2) 微支付(Micropayments)。

(3) 数字化图书馆。

这绝不是惟一的一个列表。任何包含关于 Web 资源信息的技术都应根据 RDF 模型表达它。计划是将 HTML LINK 关系转换为 RDF 属性。我们可以继续那些 RDF 明显合适的例子：

(1) 版本控制信息。

(2) 通用和专用 URI 之间的关系。

(3) 存取控制信息。

(4) 许多构件资源的复合工作中的结构化信息。

(5) 文档与其样式表之间的关系。

给定一个全球的断言语义 Web，应用于 HTML 页的搜索引擎技术可能将直接翻译成 RDF 对象的索引，而不是单词的索引。这本身将使得 Web 检索有效得多，就如 Web 是一个巨大的数据库，而不是一本大书。

当检索 HTML 页的搜索引擎发现许多搜索结果且覆盖了 Web 的很大一部分时，就会返回许多不合适的结果。对这样的搜索不存在“正确”的概念。对比而言，逻辑引擎一般能够将输出限制到正确的结果范围内，但不能够彻底搜索以构筑正确的结果。但是，已经成功的搜索引擎迫使我们重新审视这里的假设。如果未来的引擎将推理引擎和搜索引擎结合起来，那么

它将得到最好的结果。它将能够得出包含给定术语所有取值的非常完整的列表，然后使用逻辑去除那些除对解决给定问题有用项之外的所有项。人们还期待有一系列商业动机来开发能高效解决特定类型问题的引擎和算法。

4.3 语义 Web 的概貌

一种对计算机有意义的新 Web 内容形式将可能带来新的革命，这就是语义 Web。

4.3.1 语义 Web 应用的设想

当电话铃响起时，娱乐系统正在播放甲壳虫乐队的“We Can Work It Out”。Pete 接听电话时，电话向其他所有的本地设备发送了一条降低音量的消息，这些设备都有音量控制。这个电话是他的姐姐 Lucy 从医生办公室打来的，Lucy 说：“妈妈需要看病并需要进行一系列物理的治疗，我准备用我的 agent 来预定约会。”Pete 很快地同意了与她共享汽车司机。

在医生的办公室，Lucy 通过她的手持网络浏览器指示了她的语义 Web agent。Agent 立即从医生的 agent 中检索了关于她母亲的医疗信息。接着它开始尽量找到一个可行的约会时间(这些信息由站点的个人 Agent 提供)。

几分钟后，agent 给他们提供了一个计划，而 Pete 并不喜欢这个计划。他让自己的 agent 用更严格的参数(关于位置和时间)进行查找。Lucy 的 agent 完全信任 Pete 的 agent 在任务中的作用，通过提供接口认证和数据路径自动地提供帮助。

Agent 查找出一个更靠近的诊所和更早的时间计划。但是有两个警告信息：第一，Pete 将重新安排一些他的不太重要的约会。他检查了这些是什么约会。另一个问题是关于保险公司未能在物理治疗提供保险。

Lucy 对这一计划表示了同意。

4.3.2 表达意思

Pete 和 Lucy 能使用他们的 agents 来执行所有的这些任务。这要感谢的不是今天的万维网而是即将演变为明天的语义 Web。现在大多数设计的 Web 内容是由人来阅读的，不是由进行有意义操作的计算机程序来读的。计算机擅长为布局和处理分析 Web 网页。这里有一个 header，那里有到其他页的一个链接，但是总而言之，计算机没有可靠的方法处理语义：这是 Hartman 和 Strauss Physio 诊所的主页，这个连接去 Hartman 医生的履历表。

语义 Web 将把结构放进有意义的内容的网页，软件 agents 能从网页漫游到网页为用户执行复杂的任务。来到诊所的网页的 agent 不仅知道网页有关键词例如“处理，药，物理，治疗”(现在这些可以被编码)，而且知道 Hartman 医生星期一、星期三和星期五在这个诊所工作以及日期是用 yyyy-mm-dd 格式表示。它将“知道”所有这些，这些语义在网页中被编码。诊所办公室经理使用 off-the-shelf 软件根据物理治疗协会的站点上所列出的资源编写语义 Web 页。

语义 Web 不是一个分离的网，而是现有网络的扩展，能够更好地使计算机和人合作。将语义 Web 组织到现有的网络结构的第一步已经开始了。在不久的将来，随着机器变得有更好能力处理并且“理解”显示的数据，这些发展将迎来重要的新功能。

万维网的必要性质是它的普遍性。超文本链接的作用是“任何东西能连接到任何东西”。

因此，Web 技术一定不能在乱涂的草稿和正规的表现之间，在商业和学术的信息之间或在文化、语言、媒体等之中存在区别。迄今为止，Web 的飞速发展是因为文档媒体而不是作为数据和能自动被处理的信息。语义 Web 试图弥补这一点。

4.3.3 知识表现

为了使语义 Web 工作，计算机必须能够结构化信息和规则集合，计算机利用这些规则来进行自动推理。人工智能研究者在 Web 发展以前就已经研究了这样的系统。如“知识表示”系统和 Web 来临前没有实用化超文本一样，很明显它是一个好思想，并且存在一些很好的示范，但是它还没有实用化。它包含重要的应用萌芽，但是要认识到：它的完全的潜力必须要有一个单独的全球的系统。

传统型的知识表示系统典型地是集中式的，要求每个人共享普通的概念的确切定义，例如：“父母”或“车辆”。但是集中式的控制存在很大的问题，即“组合爆炸”。

这些系统通常小心地限制能被问的问题以使得计算机能可靠地回答。这个问题使人想起数学上的 Gdel 定理：任何复杂的系统也有无法回答的问题，与基本的悖论“这个句子是假的”非常相似。为了避免这样的问题，传统的知识表示系统通常有它们自己缩小的、特定的规则集合，用来对相关的数据进行推理。例如：一个家谱学系统、在家谱的一个数据库中起作用，可能包括规则“叔叔的妻子是婶婶”。

语义 Web 研究人员接受的是这样的思想：如果要实现多样性，必须要为悖论和无法回答的问题付出代价。使规则语言尽可能地具有可表达性，这样可以允许 Web 能广泛地推理。系统的表达能力使得大量可用信息和搜索引擎生成非常完整的大量资源的索引。因此，语义 Web 的挑战是提供一种语言，它能表示数据和关于数据的推理而且允许从任何现存的知识表示系统中提取规则到 Web 之上。

在 Web 上增加逻辑意味着使用规则进行推理、选择行动并且回答问题，这是目前语义 Web 任务。数学与工程复杂了这项任务。逻辑必须有足够能力描述物体的复杂性质。幸好想要表达的信息大多数沿着“一个 hex-head 螺栓是一种机器螺栓”的思想，用一点额外的词汇语言把它写在现存的语言中。

开发语义 Web 的两个重要技术是：扩展标记语言（XML）和资源描述框架（RDF）。XML 可以让每个人创建他们自己的标记，例如＜zip code＞或＜alma mater＞的隐蔽标签，这些标签注解了在一个 Web 页上的文章页或节。脚本或程序能用复杂的方法使用这些标签，但是脚本作者必须明确什么样的页用哪个标签。简言之，XML 允许用户把任意的结构加到他们的文档，而不用说明结构意味着什么。

将 RDF 表达意思编码为 3 元组，每个 3 元组与一个基本句子的主语、谓语和宾语相似。这些 3 元组能使用 XML 标签来编写。在 RDF 中，一个文档断定特别的东西（人，Web 页或无论什么）有性质（例如“是姐妹”，“是作者”）并且有某个值（另一个作者，另一个 Web 页）。使用这个结构是一个描述机器处理的大量数据的自然的方法。每个主语和宾语用一个统一的资源标识符（URI）确定，正如在网页上的链接（URL，统一资源定位符，是 URI 的最普通的类型）。谓语也被 URI 确定，它使任何人能够定义一个新概念，一个新谓语，就是为它定义的一个 URI。

RDF 三元组形成了相互关联的信息 Web。因为 RDF 使用 URI 来把这些信息编码为一个文档，URI 保证概念不仅是在一个文档中的词，而且是能联系到每个人都能在 Web 上发现的

惟一的定义。例如:想像访问许多人的信息数据库,包括他们的地址。如果想要找生活在一个特定邮政编码区的人,需要知道在每个数据库中哪个字段代表名字并且知道哪个代表邮政编码。RDF 能指定"数据库 A 的字段 5 是邮政编码类型的字段",每个术语使用 URI 而非短语。

4.3.4 本体(Ontology)

两个数据库可以使用不同的标识符,虽然事实上它们是一样的概念,例如:邮政编码。想要比较或联合两个数据库信息的一个程序必须知道这两个术语代表着一样的意思,程序必须有一个方法,无论它遇见什么数据库,都能发现这样的普通意义。

解决这个问题的一个方法是由语义 Web 的第 3 个基本的部件提供的。在哲学中,本体论是关于自然存在的一种理论,作为学科研究的理论,人工智能和 Web 研究人员有他们自己的术语,对他们来说本体论是一个文档或文件,它在正式术语之中定义了关系。对于 Web 来说,最典型的本体论是一个分类和一套推理规则。

分类定义了对象和关系的类。例如:可以将一个地址定义为一种地点类型,城市编码可以被定义仅适用于地点等。类、子类和实体之间的关系对 Web 来说是一个很有力的工具。我们能通过指定类的属性和允许子类继承属性来表达实体的大量关系。如果城市代码必须属于城市类型而且城市通常有 Web 站点,我们就能讨论与城市代码相关的站点,即使没有数据库将城市代码直接连接到 Web 站点。

在本体论中的推理规则提供了更有力的方法。一个本体论可以表示规则"如果城市代码与一个州代码相联系,并且一个地址使用那城市代码,那么那个地址与州代码相联系"。一个程序能很容易地演绎推理,例如:Cornell 大学地址,在 Ithaca,必须处于纽约州,它在美国,因此应该被格式化为美国标准。计算机并不真正"理解"这些信息,但是它现在能用对人类用户有用且有意义的方法,更有效地操作术语。

如果 Web 上有本体页,那么就能找到关于术语问题的答案。术语的意义或 Web 上使用的 XML 代码通过从页到本体论的指针定义。如果我指向的本体把 zip code 包含在地址内而你指向的本体使用 postal code;如果本体(或其他 Web 服务)能提供等价关系,那么就能解决这种混乱。本体可以包含我的 zip code 等价于你的 postal code 的信息。

当两个数据库指向地址的不同的定义时,对地址的不同的概念使用不同的 URI,而且会发现概念是完全相关的。程序能利用邮政地址的列表(在第一本体中定义的),并且为把它变成物理地址列表(第二本体中定义的)提供服务。本体提供的这种结构和语义使得对于企业家来说,提供这样的服务是非常简单的而且可以完全透明地利用。

本体可以在很多方面加强 Web 的功能。简单来说它可以改进 Web 搜索的正确性。搜索程序可以仅寻找引用一个具有精确概念的网页而不是那些使用模棱两可的关键词的网页。更高级的应用将使用本体把网页上的信息联系到相关的知识结构和推理规则。使用这种方法的一个例子是 http://www.cs.umd.edu/~ hendler。如果浏览到该页,将看到标准的网页,标题是"James.A.Hendler 博士"。能很容易地发现关于博士个人简历的链接,而且可以知道 Hendler 在 Brown 大学获得他的博士学位。然而一个计算机程序试图发现这样的信息,将不得不进行很复杂地猜测这个信息可能在个人简历中,而且要理解英语。

对于计算机来说,网页被链接到定义计算机科学部门信息的一个本体网页。例如,教授在大学工作并且他们通常有博士学位。在网页上更进一步的标记(不是由典型的浏览器显示)使用本体的概念表明 Hendler 博士的 URI 是 http://www.brown.edu/获得博士学位,这个 URI

是 Brown 大学的网页。计算机也能发现 Hendler 是特别研究计划的一个成员，有一个特别的电子邮件地址等。所有的信息很容易地被计算机处理，而且可以被用来回答问题(例如 Hendler 博士在哪里获得他的学位)，这些问题要求由搜索引擎打开的网页内容。

另外这个标记可以使开发处理复杂问题的程序更加方便，而这个问题的答案不在单一的网页上。假定你希望寻找去年在一个贸易会议上遇见的 Cook 女士，但是你不记得她的名字，只是记得她是你的顾客之一，而且她的儿子在你的母校。智能搜索程序能通过筛选所有名字是"Cook"的人的网页(回避掉所有关于 cook，cooking，Cook Island 等)，发现那些属于你公司客户列表的工作人员，而且还有关于其子女的链接，继续找下去你可以知道他们所属的学校。

4.3.5 主体(Agent)

当从各种各样的来源收集 Web 内容、处理信息并且与其他程序交换结果时，人们将真正认识到语义 Web 的力量。这样的软件主体将指数地增加，使机器可读的 Web 内容和自动化服务(包括其他的主体)变得可行。

主体作用的一个重要方面是语义 Web 的"证据"交换，它是统一语言的(表示逻辑的推理的语言，这些推理使用本体证明的规则和信息)。例如：假定 Cook 女士的联系信息可以用一种在线服务定位，你自然地想要检查，因此计算机要求服务证明它的答案，这个答案是它通过把它内部推理翻译成语义 Web 的统一语言而提供的。在你计算机中的推理引擎很容易验证这位女士确实是你正在寻求的。如果你仍然有疑问，它能给你看相关的网页。

另一个重要的特征是数字签名，它是数据的加密块，计算机和主体能使用其证实是由特定的可信来源提供的相关信息。有个过程称为服务发现，只有当一种普通的语言在让其他的主体"理解"提供的方法和如何充分利用时，这个过程才会发生。服务和主体能为它们的功能做广告，例如在类似黄页的目录写下这样的描述。

一些低层次的服务发现策略当前是可行的，例如微软的 Universal Plug and Play，它集中了不同的类型设备的连接，以及 SUN 微系统的 Jini，它试图连接服务。然而这些想法，在结构或语法的层次中存在问题而且严重地依赖功能描述的预测集合。

相反语义 Web 是更灵活的。消费者和制造商的主体能达到共享理解，通过交换本体，它提供为讨论准备的词汇。当主体发现新的本体时它们甚至可以"自己产生"新的推理能力。语义也可以使得充分利用服务更容易，这些服务仅是部分匹配请求。

典型的处理过程将包含"值链"的生成，在其中信息的组件从一个主体传递到另外一个，每个"增加值"构成终端用户请求的最后结果。为了根据需求自动地生成复杂的值链，一些主体除了利用语义 Web 外，还利用人工智能技术。但是语义 Web 将提供基础和框架，使得这样的技术更可行。

将所有的这些特征合并会得出本文开始时所展示的主体能力。Pete 和 Lucy 的主体代表了服务广告中的服务和主体的各种任务，例如：它们能使用一种可信任的服务来获取服务提供者的列表并决定哪个是可行的。服务提供者的列表是由其他的搜索服务提供的。这些活动形成的链逐步减小到小数据量信息，满足 Pete 和 Lucy 的时间表和其他要求。

语义 Web 将突破虚拟领域并且扩大到我们的物理世界。URI 能指向任何东西，包括物理的实体。这就意味着我们能使用 RDF 语言来描述例如房间电话和电视等设备。这样的设备能为它们的功能做广告——它们能做什么以及它们怎么被控制，与软件主体相似。这样的语义方法打开了令人激动的可能性的世界，它比低层次的策略如 Universal Plug and Play 更加灵

活。

现在的家庭自动化，要求小心地配置以便一起工作。设备能力和功能的语义描述将让我们用最少的人干预完成这样的自动化。一个细微的例子就是当 Pete 接听电话时，立体声降低了。不必为每个特定的设备编写程序，他可以编写这样一个函数，包括每个广告上说明有音量控制的设备，如电视、DVD 播放器，甚至放在膝上的媒体播放器，这是他今晚下班后带回家的设备。

随着开发描述设备功能能力和用户偏好标准的研究，在这个领域中已经走出了坚实的第一步。建立在 RDF 上的这个标准被称为 Composite Capability/Preference Profile（CC/PP）。开始时，它会使房间电话和其他非标准的 Web 设备描述他们的特征以便 Web 内容能立即被他们定制。接着，当我们加上完整的语言以处理本体和逻辑时，其他设备能自动地增加信息或功能。不难想像基于 Web 的微波炉要向冷冻食物制造商的 Web 站点请教理想的做饭参数。

4.3.6 知识的进化

语义 Web 不是"仅仅"执行单个任务的工具。如果适当设计，语义 Web 能在整体上帮助人类知识的进化。

人们努力在较小的活动小组的有效性和广大的需要之间进行协调。一个小组能很快而且高效地变革，但是这产生了一个小群体文化，其他人不能理解它的概念。然而大群体的协调行动是非常慢的而且需要大量的通信。

当需要一种更普通的语言时，一个必要的过程就是把所有子群体文化加在一起。通常两个群体独立地发展类似的概念，而且描述两者之间的关系会带很多好处。当相同的概念没有相同的词汇时，是允许关系通信和合作的。

语义 Web，每个概念都简单地由 URI 命名，可以使任何人用最少的努力来表达它们发明的新概念。它统一的逻辑语言将使这些概念能够不断地连接到全球 Web 上。这种结构将通过软件 agents 使人类知识和工作进行更有意义的处理，为我们的生活、工作和共同学习提供一种新的工具。

4.4 Web 的设计原则

Web 的设计原则集中了当今 IT 技术发展的智慧和经验，是多少代从事信息技术研究和开发的杰出人士的集体智慧的结晶，值得每一个系统级技术和管理人员研究、体会和把握。

1. 可维护性(Maintainability)

规约和文件以及根据它们制作的程序都必须是可以维护的。几乎没有什么数据和服务从不需要更新、移动或转换。有时一个程序可以实现这些改变，但总有一个时刻主要的改变必须由人来参与。此时事务规模如果易处理或者有一个相当清晰的结构就好多了，毕竟从最后一次看到它开始，要改变的事务已发生很长时间了，而且最初的创作者转到更好的工作岗位也已很长时间了。一个 XSL 的样式表比一个 CSS 的样式表更难改变，但即使如此，XSL 的设计者花费大量的时间用于正确选择关键字、提供注释语法和允许在任意位置编码的做法是很好的。

CSS 有许多特征可为那些维护风格的人们提供帮助：允许在逻辑单元里分裂大的样式表，把事务集中在一处，创作者期望一起改变，速记属性提供了一种方便的方式用于设置通常一起发生的单规则多属性。

另一方面，CSS不缺乏程序员在它们程序设计语言中所使用的更加强有力的特性：宏、变量、符号常量、条件表达式和变量表达式等。那是因为这些特性给有能力的用户提供了许多方便，但对于缺乏经验的用户来说这是很困难的。这是平衡的，而且对于CCS，这种平衡不同于其他事务的平衡。

2. 模块化(Modularity)

任何时候人们都仅用有限数量的概念有意识地工作。解决一个知识性问题时所用的短期记忆只能保留六到七条内容，因此为解决一个有许多变量的复杂问题，人们把问题分解成至多六块，每一块是一部分问题，必须依次轮流解决，但也可用它来代表其他的问题。当块足够小时，可以为它起一个易记住的名字称做模块。

如何把问题正确地分成块？基本上是依靠反复试验和错误，尽管大部分是基于语言技巧的直觉。

当定义模块时，人们尽力去寻找“逻辑单元”，即似乎属于同一类的事物。这个过程与逻辑无关，但却与在一个公共名字下分组事务的能力有关。因为如果用一个短名字来命名一组，或许这意味着它是一个为人所知的问题。基本上依赖于名字所喻含的意思，因为这些名字是从其他问题领域中引用的。因此有了“口令”、“样式表”、“防火墙”、“协议”，当然还有“Web”。

3. 可访问性(Accessibility)

确保易访问性的主要方法是以尽可能高的抽象程度编码数据，但注重已经存在的易存取技术也是重要的。

结构/风格两分法是前者的典型例子。W3C格式首先让用户编码为什么它是红的而不是仅编码它是红色的。在此基础上，增加红色作为一种准则。举个HMTL的例子，认真仔细的创作者不会写<font color="red">，而会写<em class="warning">，并把warning以红色显示。某些人从不用(或没见过)的这种红色显示方式至少是其他一些警告方式的代替品(如难听的声音)。

有时可以将一项完整的技术看做易访问性的特征。SVG有许多胜过光栅图像格式的其他优点，但非常重要的一点是它允许图像作为一个对象层次被分割并旋转，而不是把图像看做着色像素的集合。

制造易存取资源不需要做更多的工作吗？人们不会懒惰而删除结构和注释吧？一些人想做得好，一些人需要做好(因为制度或因为老板的公众形象)。但实际情况是适当的结构和注释后编码易于维护，易访问性有时是一种投资，最初投入成本，最后获益。在设计优良的格式中，易访问性的特征不是作为事后的想法而增加的，而是与易维护性和代码结构特征相结合的。一幅SVG图形能够被解析并且各个图像块可再次使用创建新的图像。一幅PNG图像，则相反。

HTML中增加img元素没有考虑易访问性(或者与易访问性相关的其他一些东西，如所依存的事物和标题说明)，并且被提议的fig和object选项仅因出现得晚而从未产生作用。必须从一开始就考虑易访问性特征。

4. 设备独立(Device-independency)

如果规范不依赖具体设备，那么它应当被分为依赖和不依赖设备两部分。尽管设备独立性的前提不同，它们在许多方面与易访问性相同。

像CSS一样的规格说明有点设备依赖性：制定一种字体仅在视觉媒体中有意义，在语言合成器中没有这种字体的解释。CSS事实上把设备分类并根据样式表把每一类组成一组。

HTML 不同,准确地说是因为 CSS 关注的是依赖设备的部分,HTML 是不依赖于设备。因此,可以用 HTML 做更多的事。

有人称 HTML 不具有设备独立性,因为 HTML 创作者假定读者都有一个方便的媒介用于阅读文本,如一本书,一本杂志以及激光打印机或者一个相当大的屏幕。在一个移动电话上显示 5 页的文章不会令人感到愉快。但那是 HTML 的过错吗,仅是某种内容更适合某种环境的问题吗?

在同一时间不同设备上能够使用同一资源不但令人愉快,而且一种有价值的资源也应当被设计成能够适应将来技术变化:技术像现在这样迅速地发展,一个 4 年前在最新的系统上写的文档可能已经无法读取了,因为硬件不再制造,软件制造者已经破产等。

5. 国际化(Internationality)

考虑到世界范围,Web 对于那些不会使用开发者的语言的人们来说也同样应当适用。通常开发一个仅对一两种语言适用的信息交换格式是没有太大意义的。因此如果某个规范在何处允许可读文档,它就应允许该文档可以是任何语言。文本必须是统一的字符编码标准库中的任何字符,并应该允许至少包括 UTF-8 的不同的编码方式。

不要设想文本总是水平的,从左到右并且在单词之间存在空格。并不是每个人都认为 7/1/92表示 92 年 1 月 7 日。

当你为 W3C 撰写规约时不要忘了 W3C 的规约是用英文来表达的,但多数情况下是被那些将英文作为第二语言的人们来读的。

6. 可扩展性(Extensibility)

如果一项技术恰好是第一次采用就有很好的结果,那么就没有不同的版本,没有不同的应用,也没有升级问题。但实际工作中每一项都会有版本 2 甚至版本 3。所以在设计版本 1 时最好考虑到这种情况。

可以做一些事情使得将来的扩展变得简单些,尤其考虑在保留的语法形式方面。在其中可以加入新的特性,可以使得语言在一定的范围内向前兼容。

一个简单的技巧是在文件头加入"魔术数字",其中包括版本数字,从而针对版本 1 的应用就会认识到文件不是正确的版本。这避免了数据的曲解,但这种方法是生硬的。

比如 HTML 有一条简单的用于文本文档的合理规则,即将所有未知元素中的文本看做简单的内联文本同时忽略掉未知属性。新版本的开发者有两种选择,这要看新特性注重忽略文本还是内联显示它。HTML 的文件头也有"魔术数字",但采用了较好的方法,使得浏览器在看到文件是一个很新的版本时不会放弃。

CSS有稍微复杂的方法,因为它允许用户在很多情况下包括对旧应用明确的可替换物。旧的程序将会读它们所理解的而略掉其余的,新的程序将会两者都读,如果用户正确应用,新的特性将会覆盖旧的特性。

更加复杂的方法是为用户提供一种手段来明确某种情况的近似值是否可以接受。C 语言提供了一个"#error"关键词,程序员可以用它来向编译器发出信号表明某种特性是必须的,没有它继续执行是无意义的。

7. 可学习性(Learnability)

有时候你会听说语法并不重要。并非如此,语法是相当重要的。认为语法不重要的人,在描述同一个模型可能有很多方法,但并不是所有的方法都是等同的。对于正式的、有限的语言我们必须接收来自数学方面的建议:没有比用启发式符号更帮助理解模型的了。

可学性很大程度上需要可读性。

8. 可读性(Readability)

一个符号可能太短。如果一个偶尔用的词只用一个字母表示(如"t"),那么如果没有彻底搞清这个字母的含义,当看到这个字母后,也许不得不查一查它的意思。如果一个词不是太长的话,最好用整个单词表示(如"translate")。

一个符号也可能太长。如果经常用的关键字有 20 个字母(如"shapeoutlinedata"),那么就应该用一个简短的符号来表示(如"d")。这些例子取自 SVG,但是在大部分的语言中你都能发现相似的情况。

但有时最好的猜测可能是错的。XML 的设计者认为用一个要素的全称对于打开或者关闭这个要素是有益的(<heading>...</heading>);对于使用者,他们的想法确实是十分合理的:在长的文章中用一些稀少的标志表明文章的要素,减少文章的冗余,允许用一个较短的形式(如</>)正确地加入到一个复杂的语言中去。但是 XML 现在更常用于那些标志性的数据。每行的开始标签以及紧接着的和它们同样重要的结束标签通过它们间的冗余被隐藏在基本的数据之后。

当然可以用其他的语言而不是 XML。如果资料的可读性是重要的,也许应该这么做。但是却有另一种麻烦:当习惯了 XML,不得不去想如何表达一些事情(如何避免统一的字符编码标准的特征,怎样确保语法的不含糊性,什么样的分割符用于嵌套的结构,如何分解空白等)。

一个办法是使用 XML 和一个易读的版本,可以在两者之间进行转换。对于 MathML,可以用一些熟悉的符号去编辑数学公式的工具。一旦有了这些工具,资料的可读性将变得不太重要。

因为可读性对一种语言来说是非常重要的,CSS 有它自己的语法,它可能基于 SGML,但毫无疑问 CSS 语法是很好的(XML 那时不存在)。

对于一些其他语言选择 XML 和一些其他语法的复合,当其他的一些符号如冒号(:)、分号(;)、大括号({})和括号(())代表一些意思时,人们通常更习惯。例如 SML 表示时间间隔用这样的符号表示 3:25.5,而不说<time><min>3</min><sec>25.5</sec></time>。同样 SVG 用一个紧凑的符号表示路径以使得非常好读,而不是用曾经用过的 XML 标签。但是 SMIL 和 SVG 仍然用 XML 作为路径,在这里冗长不是主要问题,虽然降低了一定的可读性,但是使制作设计者规范工作变得更容易。

相比较而言(如对效率),也有些语言在可读性方面是不重要的。PNG 是一个例子,它是二进制格式的。

9. 有效性(Efficiency)

Jakob Nielsen 认为当计算机对人们点击的反应时间小于 1 秒时,人是精力集中的,当 1 页需要多于 10 秒的时间来显示时,人们就会失去注意力。

当然,计算机的功能会更强大,带宽会不断地增加。但同时,Web 上连接了新的设备,比如移动电话和电视,它们又变得更慢。尽管它们可能落后于桌面电脑,但它们最终也是会发展的,毫无疑问会有新的应用。Web 技术的设计者们必须经常考虑到效率问题。

对于服务器来说,如果对于一个询问产生的应答非常复杂就会减慢速度,对于网络来说如果数据量太大也会减慢速度,对于客户端来说如果显示数据很困难也存在同样的问题。

当设计一种形式语言时,设计中需要保证语言的容易解析性和合理的简洁性。"Progressive Rendering"(在所有数据到达之前显示应答的效果)很有效。例如 CSS 就是采用这种方

法：文档的行一到达就立即被显示出来，高速缓存也很有效，将文档划分为可缓存和不可缓存的部分。比如 HTML 和 Xlink 允许文档按部分聚合起来，这些部分可能在其他的文档中被共享。有时可应用一些压缩工具和转换成二进制格式的工具。

在语言中提供减少冗余是很好的思想。因而 HTML 和 SVG 都有将类指派给一个元素并在一个地方定义该类的所有元素样式的技术。SVG 同时允许绘图对象在不同地方的重用：画一个 12 点的星仪需要一个点，并将该点在不同的地方显示，共显示 12 次。

直觉并不能很好地指导效率的评估。网络的吞吐量也不是线性的：由一个 TCP 包组成的消息比由两个包组成的消息快得多，但四个信息包要比两个独立的信息包快很多。通常 Java 是较慢的，但 Jigsaw(用 Java 写的 HTTP 服务器)可以与 Apache(用 C 写的 HTTP 服务器)相抗衡，因为人们认为 Java 慢的原因主要是在 Java 虚拟机的启动，可是如果运行起来 Java 是相当快的。正如 W3C 曾经研究过的，更多的效率来自根本性变化：PNG 代替 GIF 和 HTTP/1.1 代替 HTTP/1.0 有一定的作用，但更明显的是用一些带有样式表的文本代替图像。

10.二进制数据格式或文本格式(Binary or Text Format)

大多数的 W3C 规范定义了一种用于描述某种类型资源的形式化语言：HTML 描述简单的文本文件，SVG 描述矢量图形，PNG 描述光栅图像，HTTP 描述客户与服务器之间的对话以及 URL 描述一种特定资源的路径。有例外，如几个 WAI 指导方针，用于描述关于如何设计程序和规范的元规则。但大多数使用 W3C 工作的人们不得不从这项选择开始：我们创建一个二进制格式的呢，还是文本格式的？

在大多数情况下，答案是“基于文本”。因为文本格式易于建立和调试：可以用文本编辑器创建文件，可以在以后再开发一种专用的编辑器和转换器；可以检查文件，看已经发生了什么，以防程序并没有按所期望的去执行；如果在大约 50 年里说明规范偶然丢失了或难于发现，那么就有机会从观察一些可以逆向设计的文件中再次获取必要的信息。这被称为“自描述”。如果每个文件都包含说明文本，这种格式才是真正的自描述。

基于文本的格式也考虑到了以后不可预见的扩充，因为存在大量的冗余是它的一大特色。许多程序语言因此形成惯例，用结构化的注释说明事物，或者在以后的版本中获取新的关键字。尽管二进制格式有扩充机制，但仍有诸多限制。

选择基于文本的格式仅是因为 IETF 推荐，或因为某位管理人员需要 XML。

事实上二进制格式不是那么糟糕。它们在处理和传输上经常更有效。它们的典型特征是规模小。使用 ZIP，MPEG 和 JPEG 的目的是用尽可能少的二进制位。它们需要比较简单的解释器，处理速度更快。

尽管文本到二进制的转换常因某个原因使它们处理速度变慢，但基于文本的解释器增加的复杂性经常导致更多的缺陷。

缺乏二进制展开性的这种格式就那么糟糕吗？或许不是。即使文本格式在理论上是可扩展的，但实际上可能不是如此。如果这种扩展不是以任何有用的方式向下兼容，那么最好设计一种全新的格式。HTML 已经过三个版本的扩充：2.0，3.2 和 4.0，使用者没有严重的问题而 HTML 也没有取得任何进展。XHTML 是 HTML 的继承者，但它是一种新格式，不是一个扩展版本。

大多数的 W3C 格式是基于文本的，并且将延续下去，但不认为二进制格式是禁止使用的。双方都需要考虑成本。

11. 可实现性(Implementability)

某项技术要求越容易实现,越需要生产很多工具。简单的实现也意味着实现者更能创造性地提出新颖方法来应用这种技术。工具会很廉价,实际上如果开发者个人利用空闲时间来自己制作就没有花费。开发者知道修复故障是花费最高的操作之一,因而避免故障是最受重视的。可实现性来自简单性或基于明确的、存在的组件。

基于已经存在的组件意味着应用程序员熟悉的技术,例如:上下文自由语法或 XML1.0。

这并不意味着在新的或是不理解的技术上增加附属物。XSLT 可能不应该应用 XML 名字空间。应该以更常规的方式区分数据和注释。正如 HTML 应用 SGML 一样,XSLT 正面临着关于非一致性子集的改革的危险。

Jakob 认为如果要采用一个新的规约,要等它成为正式规约一年以后才是安全的。

W3C 规约至少与其他组织的标准相比不是非常复杂,但也正在开始变得复杂。为了使规约变得简化,W3C 可能不得不再次着眼于 IETF 及相互依赖性,尤其是对于首先编制基于 XML 程序的开发者来说 XML 家族的规约正变得很难处理。

一个规约存在两种实现方法并不能保证将会有更多的好处,但它给了人们启发。因此许多 W3C 工作组在将他们的工作作为"推荐"之前,会探求两种或是更多的实现方法,这是一个很好的思想。

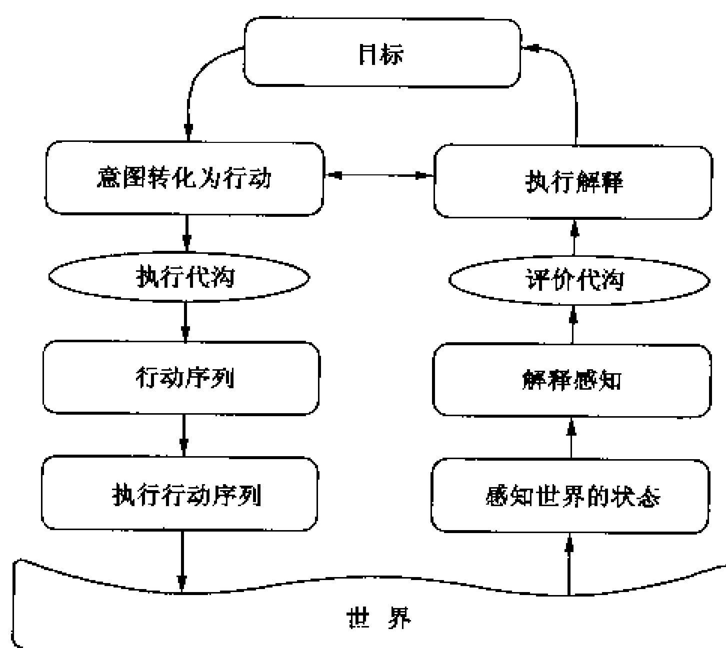


图 4.3 完成某个活动所需的七个步骤示意图

12. 简单性(Simplicity)

图 4.3 表明了一个人为了完成某个活动所要通过的七个步骤。从人的目标开始,它建立了一个意图的集合来改造世界上的某些事物。他将这些意图转换成一系列的行动然后执行它们。随后感知新世界的状态,解释他所看到的,并与他想要改变的相比较。如果目标没有达到,他可能会进行下一轮循环。

如果要用到工具,比如要改造的世界是 Web,那么意图将要被转化成行动,要应用特定的工具集合。如果此人对问题的思考方式与工具制造者不相同,转化就很困难。Norman 称此为"执行代沟"。

工具是软件和形式语言。好的程序可以掩盖一些差的语言，但如果基于语言的模型很难理解或是很复杂，则这样的程序很难编制，有时人们不知该使用哪种恰当的工具。

例如图像设计者在他们的意识中想使网页包含一部分重叠图像，在 CSS2 之前只能用 HTML 中的<table>元素。但表单元并不能重叠。一些较聪明的软件开发者可以设计生成一个界面使得他们能够将图像插到任何位置，程序将图像切成小片，每一片不重叠的图像片占据大表的一个表单元。尽管这种方法在大部分时候可以正常工作，但当其他人不通过作者的工具而试图解释如何运行或是改变图像将非常困难，这带来另一个“代沟”。

图中左边的“执行代沟”显示了用户将系统中的视图和信息映射回意识模型中的困难性。其中的问题可能是系统提供信息的不完整性和不明确性或采用了不适合用户的隐喻。上面例子中的“表”就是重叠图像的不合适的隐喻。

13. 长效性(Longevity)

在默认状态下，Web 上的文档寿命是永久性的。如果不发生写错误，文档值得保留。并不知道哪些东西不再有用或谁依赖于它，从不中断连接。

但应当以一种使它适应技术变化的方式向文档中写入内容，因为转换旧文档的代价可能太大。如何做到这一点？除非可以预测到未来，否则最好的方法是坚持标准，包括 W3C 规范和指导原则。

这增加了规范的开发者所承担的责任。他们不得不总是问自己是否规格得到提升，或者至少允许 Web 资源在 50 年的时间里可以辨读。由于开发者并不比他人更具预测未来的能力，因此它们很好地遵循了下述主要规则：

(1) 不要加入在下一个版本中不支持的特性。

(2) 如果规格说明丢失，使用易于编码的简单格式，可以是文本格式。

(3) 任何时候都尽可能不依赖于设备，也不将依赖设备的部分放入分离的模块。

如何处理那些已破坏的 URL？传统避免信息丢失的方法是在资源的内部提供冗余的元数据：一个图书馆能够为书编目以便能够根据小页的信息再次发现它。在 Web 上实际能够做得更好一点，因为可以让机器读全部或部分元数据。因此一个好的想法是在每个文档中包含标准域用于存放标识资源的信息：HTML 中的 META 和 LINK 成分，SVG 中嵌入 RDF，Java 中的@author/@version 等。

14. 向后兼容性(Backwards Compatibility)

有两种向后兼容类型：常见的一种是对以前同一版本的一个说明版本，另一种是含新技术的以前版本的新版。

没人忘记旧产品，因为没有什么比开发者正尽力去替换的老版本更让开发者熟悉的了。

但新产品的向后兼容性更不明显。在某种意义上，它是可扩展性和模块性的补充。然而可扩展性和模块性用一种将要与未来新技术共同作用的方式强调发展技术。而向后兼容性强调与已有的技术相结合的重要性。因为没有哪一项新技术凭空产生。

不仅在技术物质方面，而且在那些使用老技术用户的思维模式方面，新技术必须与老技术相兼容。引进新的技术规范总是要付出代价，但为将来的收益，必须付出。

举个 CSS 的例子：使用 HTML 的人们以及实际上的大多数文字处理者，习惯于把格式同具体的要素相联系。为制作可接受的样式表，CSS 不得不允许人们用相同的方式继续工作，而只扩充他们的选项。

作为一个反例，XSL 采用一种不同的方法进行样式表设计：人们开始以模板形式创建样

式表，然后把基本要素注入模板。这需要用户有更多的投入，但因此也提供了新的可能性。它把目标瞄向那些能够了解递归和循环、需要设计合乎潮流、将来才产生的抽象文档的、而不是已经写出具体文章的用户(例如，来自数据库的动态报告)。

新技术代替旧技术，极少是完全代替。理论上，PNG 是完全能够代替 GIF，而 XHTML 能够完全代替 HTML。但即使在那样的情况下，也需要某种形式的向后兼容性，以便于实现向新格式的机械转换。

"机械"意味着所需要的智力因素是有限的。例如：SVG 在 PNG 被用于显示图表和图形时可以部分代替 PNG。这些在 SVG 中将更好，然而，SVG 对 PNG 的向后兼容性几乎不存在。当然，在这个例子中，SVG 对这类图像所具有非常突出的优点，以至无人抱怨缺乏兼容性(但即使如此，SVG 仍提供了一种不重要的方法用于完成转换)。

Web 本身被设计成向后兼容。用于资源定位的 URL 使用 FTP 和其他的协议而不仅是 HTTP 存取资源，同样 HTTP 也可访问任意类型的文件，而不仅是 HTML。并且 HTML 允许超链接任何类型的文件(通过元件 A)或包含任何类型的文件(以 IMG 或 OBJECT 元件形式)，而不仅是 HTML 和 PNG。

15. 互操作性(Interoperability)

这个新名词是"兼容性"这一名词的变异，它只是简单地表示根据规约编写的某个程序或某个文档应该在不同的应用和不同的计算机上同样地工作。"同样"是有限定的：不能像 19 英寸彩显一样将文档确切显示在手提机的屏幕上，但可以从该文档中得到同样的感受和信息。

当比较两种典型的浏览器时问题就更清楚了：由于它们有不同的特性所以人们喜爱这个或那个，但如果网页在一个浏览器上是蓝色的，那它在另一个浏览器上就不会是红色的，更不会出现在一个上可读但在另一个上不可读的情况。

规约的开发者必须保证他们的规定可以被不同的工具实现：它应当不依赖于特定的平台，不能面向特定的编码机制，如果可能的话应当是可检验的。

通常有一些部分是可以被共同使用的，有些部分是不可以的。将规约划分为两部分，每一部分有自己的共同使用要求。

我们并不要每一个能显示静态文档(HTML)的程序都能显示动态文档(SMIL)或相反，因而 HTML 和 SMIL 是两个独立的模块。

16. 内容重用(Repurposing of Content)

在图 4.4 中大圆圈表示人类与作为媒介的 Web 网络的交互。某人有一个想法，他以机器可读的方式把它描述出来并把它放到网络上。网络把它到处传播让其他人看到，看到的人解释他所看到的东西，并由此可能成为这样的循环中一个新想法的发明人。

图中还有几个小的圆圈，每个都表示信息的修改，可能得到增强，也可能降格。上面的圆圈，反射是作者控制下的过程；下面的圆圈，不同视图由读者控制。但在这两者之间的圆圈，可由作者和读者中的任一个控制，可以被人，也可以由像 Web spiders 的程序自动控制。

还有个小圆圈描述网络能够提供给人类交流的额外价值。但如果这个圈子要运行良好的话，最初的描述一定要适合软件的处理。

17. 时机性(Timeliness)

一个新的技术要想获得成功，必须有合适的时机。如果在开发一个基本模块之前要等待很长时间，则要应用特别的解决方法，这将引起向后兼容性的问题，对那些已经采用新标准的人来说还会花费很大的代价。如果开发得太早，在需要的时候就有可能会被忘掉，或者出现更

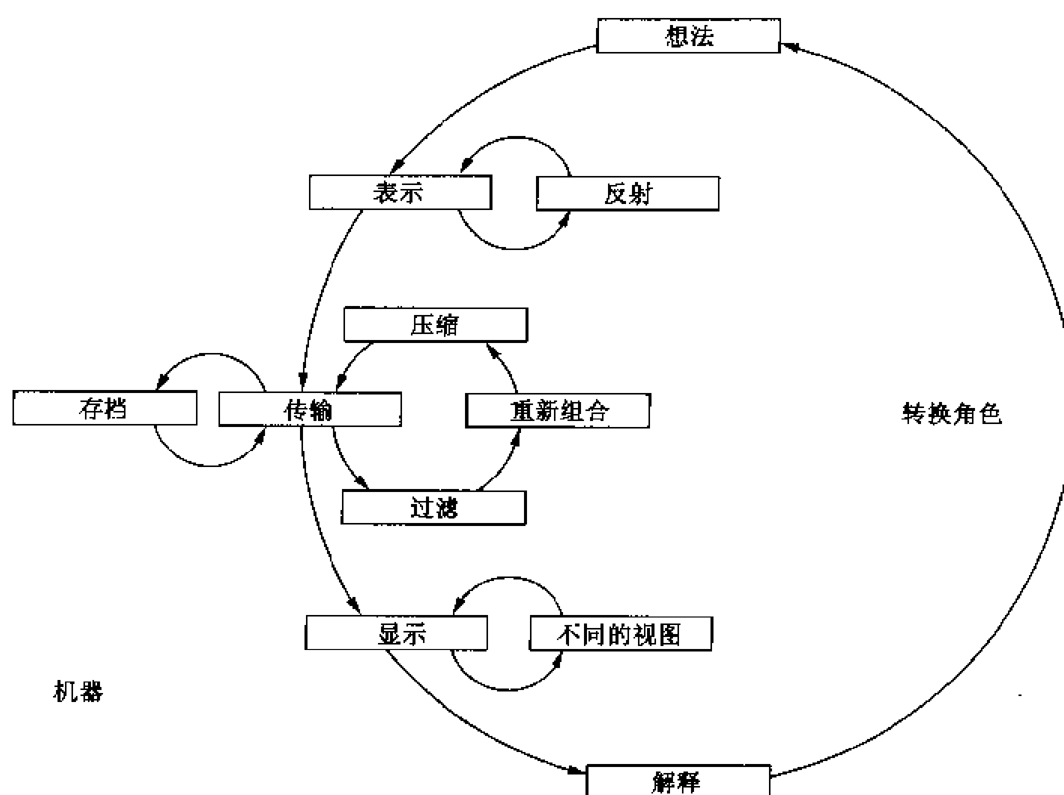


图 4.4　人类与 Web 交互过程示意图

糟糕的情况——由于中间的步骤与设想的不同而不能被应用。

什么是合适的时机？很难说。时机到来的一些征兆是人们开始问某些事物（尽管认识到他们真正问什么是一个技巧），一些人对这些事物有了一定的看法，尤其是有了很明显的解决方法。

18. 有什么用什么(Use what is there)

回顾 Web 发展的前十年，取得了革命性的进展。20 世纪 90 年代以前，HTML，HTTP 和 URL 并未出现，这不仅是因为它们形式和协议比较新颖，实际上原始的 HTTP 是一个较之 Gopher 和 FTP 等在性能上更差的协议，并且 HTML 也不是现在的样子，不能嵌入图像、表格和颜色等。

所以早期许多人都将自己的主页放到 FTP 服务器而不是 HTTP 服务器上。这些事实很好地说明了使 Web 成为可能的方法：人们并不去替换已经工作了的模块，而仅是增加那些新模块以适应已经存在的基础结构。HTML 可以在 FTP 或 Gopher 服务器上存在，浏览器可以显示纯文本或 FTP 目录列表，URL 允许各种协议等。

当前似乎到处都是 XML 和 HTTP，但造成这种观念是因为“老”的东西结合得很好以致忘记了它的存在，但 E-mail 或 Usenet、JPEG 或 MPEG 等其他 Web 上的基本部分是无可替代的。

随着时间的推移，这些技术可能有更好的解决方案。GIF 慢慢地被 PNG 代替，SMIL 代替了几种其他形式。Web 在不断的变化，任何时候都是新老技术协同工作。

各种技术发展的速度不同。试图同步发布三种或四种新的规约就已经超过了现有的能力。如果扔掉现有可工作的但不是很理想的软件，而去教会每一个人新东西是很大的资源浪

费。遗憾的是包括W3C在内的很多标准化组织会用组织内部开发的事物替换其他人开发的事物。尤其是不要试图将技术仅建立在那些没有经过检验的事物上,不管该事物听起来多么有前途,这样的冒险太大了。

19.委员会集体设计机制(Design by Committee)

几乎所有的说明都是由委员会而不是由个体设计的。W3C工作组常常有大约10到20人组成,他们为一项新技术在一起工作一年或更长时间。

"两个人知道的比一个人多"这句谚语,精确地描述了工作组存在的原因。注意的眼睛越多,检查错误越多,寻找问题解决方案越富创造性,了解过去做了什么和没做什么而积累的经验越多。

但是"由委员会设计"的一些问题仍需避免。工作组大约15个人似乎是个限度,更大的工作组倾向形成子工作组,他们会在通信上而不是在发展上浪费太多的时间。

较小的工作组步调更一致,更利于使用规格说明,但他们可能删除其他人不需要的东西。解决的办法似乎是以公共邮件发送清单的形式,围绕他们创造一个更为广泛的有兴趣的群体。

那就是W3C发展技术的方法:招用专家组成工作组,为对其感兴趣的人设置公共邮件清单。可能一些人仅对一个细节感兴趣或仅是偶然有时间探讨发展。在工作组中,他们不是妨碍了工作进程,而是通过邮件清单做出了有价值的贡献。

当然,双层体制假定委员会愿意听外部意见。需要花费一些时间去扫描邮件上的重要信息,并且根据需要回复信件。但它们中的大多数需要在部分委员会公开,并讨论它们的理由,即使它们有时是关于公司短期政策的意见。而且需要对部分公众进行对等的公开,接受正确的理由。

20.专家的意见(Expertise)

如果某项规约非常庞大,有各单独领域的专家但没有关于整体领域的专家,则这项规约过于庞大。将它分成两部分,对于每一部分设置一个工作组应该在最终可以得到更好的结果。

对整个规范,能够看清楚其中的不一致性和冗余性的专家是很多正规方法不可替代的。

21.简短性(Brevity)

阅读15页的说明与阅读385页的说明所花费的时间是不同的。典型的程序员会阅读开始20页,接着看一些例子,然后就会开始编程了。他会用自己带有逻辑性的想法来填补没看到的内容。但是说明书并不总是富有逻辑性的,很多时候它们仅是委员会成员间相互妥协的结果。

有时候通过删去解释使得规约简短而有意义,因为这样会有更多的人把它看完。

实际上从另一个角度看删去解释也算是个好主意,解释越多,它们相互矛盾的可能性就越大。把解说和注释放在一个单独的,非标准化的文档里,或者干脆放进一本书事情会更好。

但是不能删去例子,因为一个精选的例子经常能代替好几段解释,而且它们也可以作为执行新程序前检验的事例。

编程时程序员一般喜欢找出程序怎么实现的,而不是阅读文字。这是自相矛盾的:解释越少他们理解得越好。

22.最小冗余原则(Minimum Redundancy)

这部分称为"最小冗余"而不是"没有冗余"。技术是人类使用的,人类能够解决冗余。而当实际上没有冗余的时候会真正有麻烦。虽然对计算机(或者程序)来说太多的冗余不是件好事情,这意味着相同的事情要做几遍,但是它也不完全是件坏事情。比如你可以用几种不同的

方法用 HTML 把一段文字变成红色。人有不同的类型和思考方式，通过选取不同的工作方法他们能够表达出一些细微的东西，这个计算机可能做不到。

在另一方面，XML 1.0 定义了一个带有 4 个叶节点(文本字符，处理指令，空元素和实体)的树结构。它们都有些抽象，叶节点是 4 个，为什么不是 10 个或者 1 个，这个也不是很明确。

在不同的规范之间功能上的重叠部分应该保持很小，因为那样很容易导致不一致而使程序员难以适从，造成程序故障。另一方面，因为很容易保持一致，在同一规格上，做同一件事情可以有多种方法。

还用把一段文字变红的例子，CSS 提供了不同的规则和表述方式来描述"红色"，你可以不用费多大的力气就大大增强 CSS 的可用性。HTML 的把文本变成红色的属性备受抨击，因为它使用取自 CSS 的不同模式(不是使用一连串的过程)，使用的语法也有轻微区别("red" 和"#FF0000" 可以；"#F00" 就不可以)。

23. 稳定性(Stability)

由于技术的变化，未来是不可预测的，但通常需要某个特性的稳定性。重新学习怎样做某件事要有代价，编制新的程序从而通过不同的方法来完成同一个任务也是有代价的，将已经存在的文档或是其他资源转换成不同的形式也是有代价的，所以稍有变化或是没有益处的变化都是应当避免的。

第 5 章　XML 在人工智能中的应用

在这一章中，我们首先从人工智能发展历史的角度分析了信息技术、互联网络技术的发展对人工智能技术提出的新的需求，强调了知识表示在使 Web 更加“智能”的过程中的基础地位。同时，深入分析并总结了基于 XML 的知识表示的特点。然后比较详细地介绍了规则标记语言 RuleML 和 HornML，并结合实例详细讲解了语言的使用方法。

5.1　人工智能的历史与现状

人工智能是计算机科学的一个分支，其长远目标是人工智能的机器实现。经过半个多世纪的发展，人工智能产生了一大批理论成果，并获得了广泛的应用。

5.1.1　形成及第一个兴旺期（1956 年～1966 年）

一般认为，人工智能的思想萌芽可以追溯到德国著名数学家和哲学家莱布尼茨（Leibnitz，1646～1716）提出的“通用语言”设想。这一设想的要点是：建立一种通用的符号语言，用这个语言中的符号表达“思想内容”，用符号之间的形式关系表达“思想内容”之间的逻辑关系。于是，在“通用语言”中可以实现“思维的机械化”这一设想，可以看成是对人工智能的最早描述。计算机科学的创始人图灵在论文“理想计算机”中提出的图灵机模型以及 1950 年在《计算机能思维吗》一文中提出机器能够思维的论述，即图灵实验：让一个人和一台计算机分别处于两个房间里，仅通过键盘和打印机与外界的联系。由人类裁判员向房间里的人和计算机提问（比如：“你是机器还是人？”或“你是男人还是女人？”等），并通过人和计算机的回答来判断哪个房间里是人，哪个房间里是计算机。图灵认为，如果“中等程度”的裁判员不能正确地作出区分，则这样的计算机可以称为是有智能的。“图灵实验”是关于智能标准的一个明确定义。几乎在图灵上述工作的同时，冯·诺依曼从生物学角度研究了人工智能，构造了“自再生自动机”。冯·诺依曼的工作为后来人工智能中的一条研究路线（人工生命）提供了重要的基础。图灵和冯·诺依曼的上述工作以及麦克考洛和匹茨对神经元网络的数学模型的研究，构成了人工智能的初创阶段。

1956 年夏天举行的达德茅斯研讨会，被认为是人工智能作为一门独立学科正式诞生的标志。这次研讨会聚集了来自数学、信息科学、心理学、神经生理学和计算机科学等不同领域的领导者。与会者从不同角度研究了使机器具有智能的途径和方式，并决定用“人工智能”（Artificial Intelligence）一词来概括这一新的研究方向。达德茅斯研讨会开创了人工智能的第一个发展时期（兴旺期）。尔后，日本、德国等国家开始重视 AI 的研究。

在形成期和第一个兴旺期，人工智能研究出现了一些较有代表性的工作（这个时期 AI 研究的主要方向是机器翻译、定理证明、博弈等）。1953 年，美国乔治敦大学组织了第一次机器翻译的实际实验。1954 年 7 月，IBM 公司在 701 计算机上做了俄译英的公开表演。此后包括英国、苏联和中国在内的许多国家纷纷开展机器翻译的研究。利用计算机证明数学定理是又一项大胆的设想。1956 年，艾伦·纽厄尔（Newell）和西蒙（Simon）等人首先取得突破，他们编

的程序 Logic Theorist(应用启发式技术)证明了《数学原理》第二章中的三十八条定理,于 1963 年又证明了该章中的全部五十二条定理,走向了以计算机程序来模拟人类思维的道路,第一次把求解方法和问题的领域知识分离开。在 1958 年定理证明取得新成就,美籍数理逻辑学家王浩在 IBM704 计算机上以不到 5 分钟的时间就证明了《数学原理》有关命题演算的全部 220 条定理,还用了几分钟证明了该书中带等式的谓词演算的 150 条定理中的 85%,1959 年他再接再厉,仅用了 8.4 分钟就证明了以上全部定理。也是在 1959 年,IBM 公司的格伦特尔(Gelernter)研制出平面几何证明程序。博弈同样是 AI 第一个时期的研究热点,1956 年 Samuel 研制了跳棋程序,它在 1959 年击败了 Samuel 本人,又在 1962 年打败了美国一个州的跳棋冠军而荣获州级冠军。同样是在 1956 年,Selfridge 研制出第一个字符识别程序,又在 1959 年推出功能更强的模式识别程序。1960 年,McCarthy 建立了人工智能程序设计语言 LISP。从 1957 年开始,Newell、Shaw 和 Simon 等人就开始研究一种不依赖于具体领域的通用解题程序 GPS(持续研究了十年,1969 年发表最后版本)。1963 年,Green 公布了 BASEBALL(有关美国棒球赛的问答系统)。1963 年 Slagle 发表了符号积分程序 SAINT,用 86 道积分题做实验(其中 54 道选自麻省理工学院的大学考题),结果做出了其中的 84 道(1967 年 Mosis 以他的 SIN 程序再创记录,效率比 SAINT 提高了约三倍)。1965 年 Roberts 编制了可以分辨积木构造的程序,开创了计算机视觉的新领域。1965 年 Robinson 独辟蹊径,提出了与传统的自然演绎法完全不同的消解法,当时被认为是一项重大突破,掀起了研究计算机定理证明的又一高潮。

一连串的胜利使人们盲目乐观。醉心于 AI 远景的专家们做出了种种乐观的预言,1958 年,Newell 和 Simon 充满自信地说:不出十年,计算机将成为世界冠军;不出十年,计算机将要发现和证明重要的数学定理;不出十年,计算机将能谱写具有优秀作曲家水平的乐曲;不出十年,大多数心理学理论将在计算机上形成。有人甚至断言:照此趋势下去,20 世纪 80 年代将是全面实现 AI 的年代;到 2000 年,机器的智能就可以超过人。此时,兴奋的人们并未意识到初期的研究虽然很有成效,但并未抓到本质,危机正潜伏在初战告捷的欢乐中。

当人们进行了比较深入的工作后,发现人工智能研究碰到的困难比原来想像的要多得多。1965 年发明的消解法(归结原理)曾给人们带来了希望,可很快就发现了消解法的能力也非常有限,甚至无法证明某些简单事实;Samuel 的下棋程序当了州冠军之后没能进一步当上全国冠军,更不要说世界冠军;机器翻译不但没有成功,还出现了笑话;从神经生理学角度研究 AI 的人发现他们遇到了几乎是不可逾越的困难:以电子线路模拟神经元及人脑都并没有成功。这一切都说明:由于 20 世纪 50 年代人们盲目乐观和期望值过高,没有充分估计困难,没有抓到本质,因此 AI 的发展要比平时慢得多,而且遇到了更加严重的困难。20 世纪 60 年代中期至 70 年代初期 AI 受到了各种责难,进入了萧条波折期。

5.1.2 第二个兴旺期(20 世纪 70 年代中期～20 世纪 90 年代)

AI 研究者认真总结经验教训,发现仅依靠自动推理的搜索等通用问题求解手段是远远不够的。自 AI 形成以来,科学家们遵循一条明确的指导思想:研究和总结人类思维的普遍规律并用计算机模拟它的实现,创造一个万能的逻辑推理体系。而年轻一代(以斯坦福大学的年轻教授 Feigenbaum 为代表)认为,万能的逻辑推理体系根本就不可能存在,它最大的弱点是缺乏知识,主要技术(状态空间搜索技术)面临的困难是“组合爆炸”。要摆脱困境,只有大量使用知识。Newell 和 Simon 等人的认知心理学研究表明,各个领域的专家之所以在专业领域内表现出普通人难以企及的能力,主要是因为专家拥有丰富的专业知识(领域知识和经验)。1977 年

第五届国际人工智能联合会会议上，费根鲍姆（Feigenbaum）教授在一篇题为“人工智能的艺术：知识工程课题及实例研究”的特约文章中系统地阐述了专家系统的思想并提出“知识工程”的概念。这标志着人工智能进入第二个发展时期。

知识工程强调知识在问题求解中的作用，研究内容也相应地划分为三个方面：知识获取，知识表示和知识利用。知识获取研究怎样有效地获得专家知识。知识表示研究怎样将专家知识表示成在计算机内易于存储，易于使用的形式。知识利用着重研究怎样利用已得到恰当表示的专家知识去解决具体领域内的问题。人工智能的研究又有新的转折点，即从获取智能的基于能力的策略，变成了基于知识的方法研究。知识作为智能的基础开始受到重视，知识工程的方法很快渗透了 AI 各个领域，促使 AI 从实验室研究走向实际应用。

与早期研究不同，知识工程强调实际应用。主要的应用成果是各种专家系统。专家系统的核心部件包括表达专家知识和其他知识的知识库和利用知识解决问题的推理机。著名的专家系统有能用于地质勘探的 PROSECTOR 系统和可对血液传染病的诊断和治疗提供咨询的 MYCIN 系统等。作为专家系统开发中一个重要组成部分的知识获取又激发了自动知识获取——机器学习研究的深入发展。

由于理论研究的成果（例如各种表示方法的研究）和计算机软、硬件的飞速发展，各种专家系统、自然语言处理系统等 AI 实用系统商业化进入市场并产生较大的经济效益和社会效益，展示了人工智能应用的广阔前景。在这期间，分布式人工智能（DAI）的研究也受到各国科学家的重视，并投入大量的人力进行研究。

但是在兴旺发展中也不是事事顺心。十年过去了，日本的第五代机计划未能达到预期效果而不了了之。20 世纪 80 年代中、后期，人们想研究通用的智能机器或专家系统的设想开始出现危机，并产生了几个问题。一是智能系统的实时性以及与环境的交互性不尽人意，感知问题很不容易解决，声音、图像和文字信息等多媒体信息处理也是个问题，而要模拟人的直觉、顿悟、灵感等智能就更难了；二是人工智能问题在规模扩大后有了新问题，例如专家系统走向一般化时出现了问题，问题不在存储量和检索速度，而在于专家系统的专用领域有质的变化，目标判断（属于哪个领域的问题）要求更高层的知识、常识、推理知识、通用概念和理论等；三是推理问题，常识的形式化问题没有解决，常用的一阶谓词推理与常识推理有较大差别。AI 科学工作者开始再次反思反省，意识到：目前做通用智能机器或系统是不可能的；对一阶谓词推理的局限性要有正确估计；人工智能的学习问题很重要但很不容易解决。

5.1.3 第三个发展时期（20 世纪 90 年代末至今）

20 世纪 90 年代以来，随着计算机网络的普及，特别是 Internet 的出现，各种计算机技术包括人工智能技术的广泛应用推动着人机关系的重大变化。人机关系的这一转变将引起社会生产方式和生活方式的巨大变化，同时也向人工智能乃至整个信息技术提出了新的课题。这促使人工智能进入第三个发展时期。

在这个新的发展时期中，人工智能面临一系列新的应用需求。首先是需要提供强有力的技术手段，以支持分布式协同工作方式。现代生产是一种社会化大生产，来自不同专业的工作者在不同或相同的时间、地点从事着同一任务的不同子任务。这要求计算机不仅为每一项子任务提供辅助和支持，更需要为子任务之间的协调提供辅助和支持。由于各个子任务在很大程度上可以独立地进行，子任务之间的关系必然呈现出动态变化和难以预测的特点。于是子任务之间的协调（即对分布协同工作的支持）向人工智能乃至整个信息技术以及基础理论提出

了巨大的挑战。

其次,网络化推进了信息化,使原本分散孤立的数据库形成一个互联的整体,即一个共同的信息空间。尽管现有的浏览器和搜索引擎(如 Yahoo!)为用户在网上查找信息提供了必要的帮助,但这种帮助是远远不够的,“信息过载”与“信息迷失”状况日益严重。更强大的智能型信息服务工具已成为广大用户的迫切需要。另一方面,信息空间对人类的价值不仅在于单独的信息条目(比如某厂家生产出了某一新产品的信息),还在于一大类信息中隐藏着的普遍性知识(比如某个行业供求关系的变化趋势)。于是数据中的知识发现也成为一项迫切的研究课题。

机器人始终是现代工业的迫切需求。随着机器人技术的发展,研究重点已经转向能在动态、不可预测环境中独立工作的自主机器人以及能与其他机器人(包括人)协作的机器人。显然,这种机器人之间的合作可以看成是物理世界中的分布式协同工作,因而包括相同的理论和技术问题。

由此可见,人工智能在第三发展时期的突出特点是研究能够在动态、不可预测环境中自主地协调工作的计算机系统,这种系统被称为主体(Agent)。目前有关研究围绕着主体的理论、主体的体系结构和主体语言三个方面展开,并已产生一系列重要的新思想、新理论、新方法和新技术。在这一研究中,人工智能呈现出一种与软件工程、分布式计算以及通讯技术相互融合的趋势。主体研究的应用不限于生产和工作,还深入到人们的学习和娱乐等各个方面。主体与虚拟现实相结合而产生的虚拟训练系统,可以使学生在不实际操纵飞机的情况下学习飞行的基本技能,也可“享受”实战的“滋味”。

综观人工智能自新创以来的发展历程,可以看出它始终遵循的基本思路。强调人类智能的人工实现而不是单纯的模拟,以便尽可能地为人类的实际需要服务。强调多学科的交叉结合。从新创开始,数学、信息科学、生物学、心理学、生理学、生态学以及非线性科学等越来越多的新生学科被融入到人工智能的研究之中。从不满足于既有成果,已经取得成功的研究课题往往被“驱逐”出人工智能的视野。

5.2 人工智能的研究领域

人工智能有广泛的研究领域和潜在的广阔的应用前景。

人工智能核心课题可概括为:

(1) 知识的模型化和表示方法。

(2) 启发式搜索理论。

(3) 各种推理方法(演绎推理、常识性推理、归纳推理、规划等)。

(4) 人工智能系统结构(产生式系统、分布式人工智能、多主体系统等)。

(5) 人工智能程序设计语言(LISP,PROLOG,PLANNER 等)。

人工智能的主要应用领域为:

(1) 专家咨询系统(Expert Consulting System)。

(2) 自然语言理解(Natural Language Understanding)和机器翻译(Machine Translation)。

(3) 数据库的智能检索(Intelligent Retrieval from Database)。

(4) 定理证明(Theorem Proving)。

(5) 博弈游戏和决策(Game Playing and Decision Making)。

(6) 机器人学(Robotics)。

(7) 自动程序设计(Automatic Programming)。

(8) 感知问题(Perception Problems):视觉、听觉、嗅觉、触觉。

(9) 组合和调度问题(Combination and Scheduling)。

(10) 分布式人工智能(Distributed Artificial Intelligence, DAI)和多 agent 系统(Multi-Agent System)。

(11) 模式识别(Pattern Recognition)。

(12) 人工神经网络(Artificial Neural Network)。

5.3 XML 在人工智能中的应用

AI 在因特网时代获得了前所未有的发展机遇,Web 环境下的智能信息技术成为推动 AI 在网络环境中发展的一大动力。几乎与 WWW 发展同步,让 Web 更加"智能"的努力从来就没有停止过,WWW 创建者 Tim Berners-Lee 提出了称做"Semantic Web"的新一代 Web,其基本思想就是通过在 Web 信息的创作和发布中嵌入机器可阅读的代表某类知识的标注,使 Web 上的数据不仅能够被机器用于显示,而且能够被机器所理解,从而能提高信息服务的质量,并开拓各种崭新的智能化的信息服务。如果进一步将这些体现了数据与应用之间联系的知识以对用户透明的方式嵌入各种不同的信息源,则 Web 页面、数据库、程序、模块和探测设备将通过能够处理这种信息表示方法的主体连结起来,相互之间能够理解和协作。如果能够成为现实,那么这种方法将产生革命性的力量,并改变人类与信息交互的方式。AI 和其他领域的研究者也分别从"学习"的角度和信息表示的角度研究了这个问题。在研究过程中,人们逐渐认识到知识表示在使 Web 更加"智能"的过程中的基础地位。

作为通用标记标准(SGML 的一个子集),XML 的设计目标是使 SGML 像 HTML 一样能够通过 Web 发送、接收与处理。事实上,XML 技术为在因特网环境下的应用奠定了基础,使人工智能各个方面,特别是知识工程方面的研究成果能够在 Web 环境中广泛应用。基于 XML 的标记语言能够为分布于网上的知识提供一种统一的存储(Storage)和交换(Interchange)格式,真正实现因特网上知识交互、重用和共享。将 XML 技术运用于知识的表示出现了一门新的技术——知识标记技术(Knowledge Markup Techniques),并且已经成为国外大学的一门课程。

XML 可以提供构造网上知识库的合适的体系结构(Infrastructure)。因为基于 XML 的知识表示系统具有以下优点:

(1) 语法独立,通过 XML 提供统一的语法表示和存储格式。

(2) 可扩充性,可以通过对底层 DTD 或 Schema 的扩展增加新的知识表示能力。

(3) 可综合多种知识表示方法,可用相同的 XML 语言重写多种传统知识表示方法。

(4) 可以对不同信息源的信息进行集成,并形成统一的文档。

(5) 可以实现数据的结构化,允许在不同企业间进行知识交换,提供不同 AI 应用之间知识库的交换(Interchange);提供知识库与数据库、应用系统等之间互换(Exchange),甚至可以对 XML 表示的 AI 源程序进行编译或转换。

(6) 标准化,XML 是 W3C 确定的 Internet 上的标准数据格式。采用 XML 的知识表示可以在世界范围内,定义标准化的、公用的具备自我描述功能的数据;非常容易地通过企业信息门户(EIP)向外发布,达到知识共享与交换的目的。

研究基于XML的知识表示系统，首先要建立知识表示系统的模型，再采用XML语法描述该模型。XML允许作者创建自己的标记，这些标记带有一定的语义信息（知识）。但是从计算的角度来看，仅是作者自己创建对于作者本人而言具有一定含义的标记还不能使计算机理解其中的含义，即不知道标记中蕴含的概念和其他概念的关系。要做到这一点，就必须以某种方式将领域内涉及的主要概念，概念之间的关系、领域内概念满足的公理通过某种方式明确地表示，并且在知识的表示中体现这种背景。XML本质上是一套用来设计各式各样标注语言的准则，即meta-language（元语言），这使我们可以根据需要对知识表示的方式进行扩展。

针对不同的知识形式，出现了多种知识标记语言，如：AIML(Artificial Intelligence ML)、HornML(Horn ML)、RuleML(Rule ML)、DAML(DARPA Agent ML)、PMML(Predictive Modal ML)、XSLT(Extensible Stylesheet Language Transformations)、CBML(Case Based ML)、DLML(Description Logics ML)。本章主要介绍两种典型的知识标记语言：HornML和RuleML。

5.4 规则标记语言——RuleML(Rule Markup Languages)

随着因特网应用的进一步深入，推理规则已运用于电子商务标记，同时规则作为语义Web的一个重要设计问题被提出来，导致Web上的规则表示成为一种主流课题；而且由于智能主体和基于知识的系统的AI外壳都需要实现Web上知识交换，规则仍将在智能主体和基于知识的系统中扮演重要角色。

RuleML是2000年8月～9月在澳大利亚Melbourne召开的PRICAI(The Sixth Pacific Rim International Conference on Artificial Intelligence)发起的，因为世界上的许多Web研究人员和从业者均认为有必要提出一个基于XML的规则标记语言，其发起者主要是德国DFKI的Benjamin N. Grosof和HaroldBoloy以及美国Nisus Inc.的Said Tabet。他们为此成立了一个国际组织——规则标记发起者(Rule Markup Initiative)，目的是对基于XML的推理规则（包括前向规则与后向规则）进行标准化，实现基于Web的规则存储、交换、检索和使用。

规则标记发起者最先完成的工作是定义了一种实现参与者之间互操作的规则标记语言(Rule Markup Language，简称RuleML)，另外用Java语言开发了规则引擎。规则标记论坛组织还计划于2001年10月在奥地利的Vienna召开电子商务第三次国际会议。

RuleML大量吸收了已有的基于XML的规则标记语言，如BRML (Business Rules Markup Language, IBM)、RFML (Relational-Functional Markup Language, DFKI), XRML (Extensible Rule Markup Language, KAIST) 的设计方法与准则，并于2000年11月10日发布到了因特网上。RuleML允许在XML中定义前向规则和后向规则实现：

(1) 演绎(Deduction)。

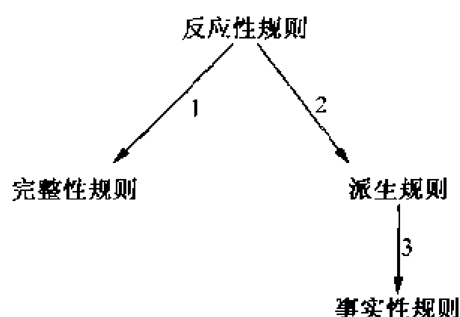
(2) 重写(Rewriting)。

(3) 进一步的推理-转换任务。

5.4.1 RuleML基本概念

1. 规则

RuleML支持4种规则表示，如反应性规则（事件—条件—动作规则）、完整性规则（一致性维护规则）、派生规则（隐含推理规则）、事实性规则（前提始终为真的规则），构成规则的层次结构。到目前为止，研究重点主要是派生规则与事实。4种规则构成的规则树如下：



4 种规则语法如下：

反应规则：<rule> <-body> <and> prem1 ... premN </and> </-body> <-head> action </-head> </rule>

完整性约束：<ic> <-body> <and> prem1 ... premN </and> </-body> </ic>

可以转换成如下反应规则形式：<rule> <-body> <and> prem1 ... premN </and> </-body> <-head> <signal> inconsistency </signal> </-head> </rule>

派生规则：<imp> <-head> conc </-head> <-body> <and> prem1 ... premN </and> </-body> </imp>

可以转换成如下反应规则形式：<rule> <-body> <and> prem1 ... premN </and> </-body> <-head> <assert> conc </assert> </ head> </rule>

事实：<fact> <-head> conc </-head> </fact>

可以转换成如下反应规则形式：<imp> <-head> conc </-head> <-body> <and> </and> </-body> </imp>

其中：

Premi 表示第 i 个前提；

<-head> ... </-head>表示规则头-head(then 部分,或结论)；

<-body> ... </-body>表示规则体-body(if 部分,或前提)；

<var> ... </var> 表示规则的变量；

conc 表示结论 conclusion。

下面是一种 RuleML 子语言 Datalog RuleML Sublanguage 的 DTD：

```
<! --An XML DTD for a Datalog RuleML Sublanguage: Monolith Version -->
<! -- ELEMENT Declarations -->
<! -- 'rulebase' root element uses 'imp' rules and 'fact' assertions as top-level elements -->
<! ELEMENT rulebase ((imp | fact) * )>
<! -- 'imp' rules are usable as general implications on the top-level -->
<! -- 'imp' element uses a conclusion role -head followed by a premise role -body, or equivalently -->
<! -- (since roles constitute unordered elements), uses a premise role -body followed by a conclusion role
        -head -->
<! -- "<imp>-head -body</imp>" stands for "-head is implied by -body", i.e., "-head is true is im-
        plied by -body is true", or equivalently, -->
<! -- "<imp>-body -head</imp>" stands for "-body implies -head", i.e., "-body is true implies -head
        is true" -->
<! ELEMENT imp ((-head, -body) | (-body, -head))>
<! -- 'fact' assertions are usable as degenerate rules on the top-level -->
```

```
<! - - 'fact' element uses just a conclusion role -head - - >
<! -   "<fact>-head</fact>" stands for "-head is implied by true", i.e., "-head"is true" - - >
<! ELEMENT fact (-head) >
<! - - -head role is usable within 'imp' rules and 'fact' assertions - - >
<! - - -body role is usable within 'imp' rules     - - >
<! - - -head uses an atomic formula - - >
<! - - -body uses an atomic formula or an 'and' - - >
<! ELEMENT -head (atom)>
<! ELEMENT -body (atom | and)>
<! - - an 'and' is usable within -body's - - >
<! - - 'and' uses zero or more atomic formulas - - >
<! - - "<and>atom</and>" is equivalent to "atom" - - >
<! - - "<and></and>" is equivalent to "true" - - >
<! ELEMENT and (atom * )>
<! - - atomic formulas are usable within -head's, -body's, and 'and's - - >
<! - - atom element uses an: - - >
<! - - -opr ("operator of relations") role followed by a sequence of zero or more arguments, or similarly -
  - >
<! - - (since roles constitute unordered elements, and the zero - argument case must not cause ambigui-
  ty), - - >
<! - - a sequence of one or more arguments followed by an -opr role - - >
<! - - the arguments may be ind(ividual)s or var(iable)s - - >
<! ELEMENT atom ((-opr, (ind | var) * ) | ((ind | var) + , -opr))>
<! - - -opr is usable within atoms - - >
<! - - -opr uses rel(ation) symbol - - >
<! ELEMENT -opr (rel)>
<! - - there is one kind of fixed argument - - >
<! - - individual constant, as in predicate logic - - >
<! ELEMENT ind (#PCDATA)>
<! - - there is one kind of variable argument - - >
<! - - logical variable, as in logic programming - - >
<! ELEMENT var (#PCDATA)>
<! - - there are only fixed (first - order) relations - - >
<! - - relation or predicate symbol - - >
<! ELEMENT rel (#PCDATA)>
```

2. 规则库

一组有序或无序的规则构成规则库(或包)。规则库可以用标签<rulebase> … </rulebase>进行标记,另外可以指定规则库的一些属性信息,如规则库名、版本号等。

下面是含有4条规则的规则库的例子,其中前两条规则是蕴含规则,后两条是事实。规则库如下:

第一条规则(用英语描述):a person owns an object if that person buys the object from a merchant and the person keeps the object.

第二条规则:a person buys an object from a merchant if the merchant sells the object to the

person.

第三条规则:John sells XMLBible to Mary.

第四条规则:Mary keeps XMLBible.

用 RuleML 表示规则库如下:

```
<rulebase>
<imp>
  <_head>
    <atom>
      <_opr><rel>own</rel></_opr>
      <var>person</var>
      <var>object</var>
    </atom>
  </_head>
  <_body>
    <! -- explicit 'and' -->
    <and>
      <atom>
        <_opr><rel>buy</rel></_opr>
        <var>person</var>
        <var>merchant</var>
        <var>object</var>
      </atom>
      <atom>
        <_opr><rel>keep</rel></_opr>
        <var>person</var>
        <var>object</var>
      </atom>
    </and>
  </_body>
</imp>
<imp>
  <_head>
    <atom>
      <_opr><rel>buy</rel></_opr>
      <var>person</var>
      <var>merchant</var>
      <var>object</var>
    </atom>
  </_head>
  <_body>
    <atom>
      <_opr><rel>sell</rel></_opr>
      <var>merchant</var>
```

```
        <var>person</var>
        <var>object</var>
      </atom>
    </-body>
  </imp>
  <fact>
    <-head>
      <atom>
        <-opr><rel>sell</rel></-opr>
        <ind>John</ind>
        <ind>Mary</ind>
        <ind>XMLBible</ind>
      </atom>
    </-head>
  </fact>
  <fact>
    <-head>
      <atom>
        <-opr><rel>keep</rel></-opr>
        <ind>Mary</ind>
        <ind>XMLBible</ind>
      </atom>
    </-bead>
  </fact>
</rulebase>
```

3. 规则的使用

定义了规则以及规则库以后，规则如何使用呢？我们可以定义如下标记：

<iterate> rulebase-to-be-iterated </iterate> 表示所要使用的规则库。

<goal> query-to-be-proved </goal> 表示所要求解的目标。

5.4.2 RuleML 实例

1. 前向规则符号的例子

一个用 XHTML 表示的超文本如下：

```
<p>If you want to review rule principles, you may look at
      <a href="http://www.cs.brandeis.edu/..."></a>
</p>
```

转换成带条件/结论的用 XHTML 表示的 RuleML 标记如下：

```
<rule>
  <prem>
      <p>You want to review rule principles</p>
  </prem>
  <conc>
```

```
      <p>You may look at <a href="http://www.cs.brandeis.edu/..."></a>
    </p>
  </conc>
</rule>
```

2. 后向规则符号的例子

另外一个格式化的 RuleML 标记的例子如下：

```
<if>
  <atom>
        <rel>may look at</rel>
    <var>you</var>
        <ur label="Rule-Based Systems">http://www.cs.brandeis.edu/...</ur>
  </atom>
  <atom>
        <rel>want to review</rel>
    <var>you</var>
        <ind>rule principles</ind>
  </atom>
</if>
```

3.Datalog DTD 描述的 RuleML 元素：Clocksin/Mellish 示例 Prolog 子句

有如下规则：likes(john, X) :- likes(X, wine)

用 RuleML 表示如下：

```
<if>
  <atom>
    <rel>likes</rel>
    <ind>John</ind>
    <var>x</var>
  </atom>
  <atom>
    <rel>likes</rel>
    <var>x</var>
    <ind>wine</ind>
  </atom>
</if>
```

有如下事实 likes(mary,wine)，用 RuleML 表示如下：

```
<if>
  <atom>
    <rel>likes</rel>
    <ind>Mary</ind>
    <ind>wine</ind>
  </atom>
  <and/>
```

```
</if>
```

由于 and 标记元素为空,即该规则的条件为真,也就是说该规则是一个事实。

4.UR-Hornlog DTD 描述的 RuleML 元素：对所提出的 W3C 页面的授权规则

假定允许任何人进行注册,即授权规则为：Any person (x) may register;用 RuleML 表示如下：

```
<if>
  <atom>
    <rel>may register</rel>
    <var>x</var>
  </atom>
  <and>
  <atom>
    <rel>person</rel>
    <var>x</var>
  </atom>
  </and>
</if>
```

假定授权规则为：Any person (x) who was some time in the last 2 monthsan employee of an organization (y) may register.用 RuleML 表示如下：

```
<if>
  <atom>
    <rel>may register</rel>
    <var>x</var>
  </atom>
  <and>
    <atom>
      <rel>person</rel>
      <var>x</var>
    </atom>
    <atom>
      <rel>organization</rel>
      <var>y</var>
    </atom>
<atom>
      <rel>employee in</rel>
      <var>x</var>
      <var>y</var>
      <cterm>
        <ctor>last</ctor>
        <cterm>
          <ctor>month</ctor>
          <ind>2</ind>
```

```
          </cterm>
        </cterm>
      </atom>
  </and>
  </if>
```

假定授权规则为：Any person (x) who was some time in the last 2 months an employee of an organization (y) which was some time in the last 2 months a W3C member may register. 用 RuleML 表示如下：

```
  <if>
    <atom>
      <rel>may register</rel>
      <var>x</var>
    </atom>
    <and>
      <atom>
        <rel>person</rel>
        <var>x</var>
      </atom>
      <atom>
        <rel>organization</rel>
        <var>y</var>
      </atom>
  <atom>
        <rel>employee in</rel>
        <var>x</var>
        <var>y</var>
        <cterm>
          <ctor>last</ctor>
          <cterm>
            <ctor>month</ctor>
            <ind>2</ind>
          </cterm>
        </cterm>
      </atom>
  <atom>
        <rel>member in</rel>
        <var>y</var>
        <ur label="W3C">http://www.w3.org/</ur>
        <cterm>
          <ctor>last</ctor>
          <cterm>
            <ctor>month</ctor>
            <ind>2</ind>
```

```
        </cterm>
      </cterm>
    </atom>
  </and>
</if>
```

5. UR-Equalog DTD 描述的 RuleML 元素：URI 扩充表达式

假设有如下 URI 扩充表达式：uriexp(daml) = http://www.daml.org/。用 RuleML 表示如下：

```
<if>
  <eq>
    <nano>
      <fun>uriexp</fun>
      <ind>daml</ind>
    </nano>
    <ur>http://www.daml.org/</ur>
  </eq>
  <and/>
</if>
```

由于 and 标记元素为空，即该规则的条件为真，也就是说该规则是无条件等式。

另有如下 URI 扩充表达式：uriexp(oil) = http://www.ontoknowledge.org/oil/。用 RuleML 表示如下：

```
<if>
  <eq>
    <nano>
      <fun>uriexp</fun>
      <ind>oil</ind>
    </nano>
    <ur>http://www.ontoknowledge.org/oil/</ur>
  </eq>
  <and/>
</if>
```

5.5 HornML(Horn Logic Markup Languages)

Horn 是人工智能和计算机科学中一种特殊类型的子句。一个 Horn 子句就是至多只有一个肯定文字的子句。HornML 是 XML 在 AI 领域的一个具体分支，是针对 Horn 类型子句的一种 XML 子集。

1. Herbrand 项(Terms)：单个常量、变量、平行结构(flat ground structures)

在 XML 中，表示 Herbrand 项采用<ind>和<struc>元素(有时还需要<var>元素)：

(1) 单个常量。如 Britain 和 France 之间的海底隧道 channel-tunnel(常量)表示为：<ind

>channel-tunnel</ind>。

(2) 平行结构。如 undersea-connection(britain,france)表示为一个内嵌<constructor>元素、带有一些参数条件的<struc>元素。

```
<struc>
  <constructor>undersea-connection</constructor>
  <ind>britain</ind>
  <ind>france</ind>
</struc>
```

(3) 嵌套结构。如嵌套结构 service-tunnel(

undersea-connection(britain,surrounded-country(belgium,luxembourg,germany,switzerland,italy,spain)))

用 XML 表示为:

```
<struc>
  <constructor>service-tunnel</constructor>
  <struc>
     <constructor>undersea-connection</constructor>
       <inc>britain</ind>
       <struc>
         <constructor>sorrounded-country</constructor>
         <ind> belgium </ind>
         <ind> luxembourg </ind>
         <ind> germany </ind>
         <ird> switzerland </ind>
         <ind> italy </ind>
         <ind> spain </ind>
       </struc>
     </struc>
  </struc>
```

关于标签(Tag)与类型(Type)的讨论:

(1) 在 prolog 中,符号(Symbol)是模糊的、孤立的,其真实意义不十分清楚;如:channel-tunnel 是一个独立的变量,而 service-tunnel 是一个构造器(Constructor)。

(2) XML 作为一个强类型的逻辑程序设计语言,符号的含义是清楚的。

(3) 上面的例子已给出了 XML——自描述语言的优点;当然,缺点也很明显,需要较多的程序空间。

2.Horn 子句:*关系符号应用*(Relation Symbol Applications)

在 XML 中谓词(Predicate)或关系符号(Relation Symbol)用<Relator>元素描述。

例如:谓词 Travel 描述如下:

```
<relator>travel</relator>
```

关系符号应用的项(terms) 用<relation>元素进行标识。

例如:应用 travel(john,channel-tunnel)用 XML 描述如下:

```
<relationship>
```

```
    <relator>travel</relator>
    <ind>john</ind>
    <ind>channel-tunnel</ind>
  </relationship>
```

3. Horn 子句：事实(Facts)

Horn 子句的事实可以用<hn>元素(Element)进行声明，其中包含<relationship>元素作为<hn>的子元素。

例：prolog 的一个事实是：travel(john，channel-tunnel)

用 XML 表示如下：

```
<hn>
  <relationship>
    <relator>travel</relator>
    <ind>john</ind>
    <ind>channel-tunnel</ind>
  </relationship>
</hn>
```

4. Horn 子句：规则(Rules)

Horn 子句的规则可以用<hn>元素(Element)进行声明，其中至少包含两个<relationship>元素，一个<relationship>元素在前，其他<relationship>元素在后。

例：prolog 的一个规则是：

travel(someone，channel-tunnel)：-carry(eurostar，someone).

用 XML 表示如下：

```
<hn>
<relationship>
  <relator>travel</relator>
  <var>someone</var>
  <ind>channel-tunnel</ind>
</relationship>
<relationship>
  <relator>carry</relator>
  <ind>eurostar</ind>
  <var>someone</var>
</relationship>
</hn>
```

5. 标识符(ID)及标识符交叉引用(IDREF)

属性类型的标识符 ID 和 IDREF 可以对元素进行命名与引用。

ID 属性值必须惟一标识一个元素，而 IDREF 属性值必须是一个元素的名称。

例：对一个 Horn 子句进行命名。

```
<hn id='john-channel'>
  <relationship>
```

```
    <relator>travel</relator>
    <ind>john</ind>
    <ind>channel-tunnel</ind>
  </relationship>
</hn>
```

对 Horn 子句命名后就可以对其引用。一个 Horn 子句可以引用另外的 Horn 子句。

例如下面典型 prolog 形式的事实:belief(mary,travel(john,channel-tunnel)),用 XML 描述时,可以引用前面定义的 john-channel 子句,这样,该事实用 XML 描述如下:

```
<hn>
  <ralationship>
    <relator>belief</relator>
    <ind>mary</ind>
    <prop idref='john-channel'/>
  </relationship>
</hn>
```

其中:belief 操作符的参数 travel(john,channel-tunnel)用<prop idref='john-channel'/>进行表示(在 XML 中,空的元素<tag...></tag>缩写成<tag.../>)。

通过对 Horn 子句进行命名及引用,可以实现共享。

6. 文档类型声明 DTD:元素派生树(Derivation Trees)

XML 的文档类型声明最初仅仅是用来描述无类型(non-attributed)元素的。

(1) 对非终端来说:DTD 描述平常的、上下文无关文法表示的 EBNF 符号。

(2) 对终端来说:常常任意地置换基本字母表('PCDATA'),而不是采用固定的终端序列。

7. Horn 逻辑

Horn 逻辑(Logic)的文档类型声明为一个具有零个或多个 Horn 子句(hn *)的知识库 kb。如:

```
<! ELEMENT kb            (hn * )>
<! ELEMENT hn            (relationship,relationship * )>
<! ELEMENT relationship  (relator,(ind/var/stru) * )>
<! ELEMENT struc         (constructor,(ind/var/struc) * )>
<! ELEMENT relator       (#PCDATA)>
<! ELEMENT constructor   (#PCDATA)>
<! ELEMENT ind           (#PCDATA)>
<! ELEMENT var           (#PCDATA)>
```

通过<kb>与</kb>符号对将派生子句括起来。即知识库 kb 表示如下:

```
<kb>hn * </kb>
......
<kb><hn>relationship relationship</hn></kb>
......
<kb>
  <hn>
```

```
    <relationship>
    <relator>#PCDATA</relator><var>#PCDATA</var><ind>#PCDATA</ind>
    </relationship>
    <relationship>
    <relator>#PCDATA</relator><var>#PCDATA</var><ind>#PCDATA</ind>
    </relationship>
  </hn>
</kb>
```

例:prolog 的一个知识库 kb 包含一条规则:

```
travel(someone,channel-tunnel):-carry(eurostar,someone).
```

用 XML 表示如下:

```
<kb>
    <hn>
      <relationship>
        <relator>travel</relator>
        <var>someone</var>
        <ind>channel-tunnel</ind>
      </relationship>
      <relationship>
        <relator>carry</relator>
        <ind>eurostar</ind>
        <var>someone</var>
      </relationship>
    </hn>
</kb>
```

8. 用 XML 符号实现 Horn 查询(Horn Queries in XML Notation)

假设有如下事实(carry(eurostar,fred)):

```
<hn>
    <relationship>
      <relator>carry</relator>
      <ind>eurostar</ind>
      <ind>fred</ind>
    </relationship>
</hn>
```

Horn 逻辑解释器可以回答如下的查询(carry(eurostar,Someone)):

```
<relationship>
   <relator>carry</relator>
   <ind>eurostar</ind>
   <var>someone</var>
</relationship>
```

通过提问(Someone):

```
<var>someone</var>
```

回答(fred):

```
<ind>fred</ind>
```

9. 用 XML-QL 实现的 Horn 查询

假设事实扩展了 prem(ises)/arity/pos(ition) 属性,如:

```
<hn prem='0'>
  <relationship arity='2'>
    <relator>carry</relator>
    <ind pos='1'>eurostar</ind>
    <ind pos='2'>fred</ind>
  </relationship>
</hn>
```

则用 XML-QL 实现查询如下:

```
WHERE
  <hn prem='0'>
    <relationship arity='2'>
      <relator>carry</relator>
      <ind pos='1'>eurostar</ind>
      <ind pos='2'> $ someone</ind>
    </relationship>
  </hn>
CONSTRUCT
  <binding> <someone> $ someone</someone> </binding>
```

10. 用 XML 符号实现 Horn 推理

假定有如下基本事实:carry(eurostar, fred),利用如下的规则:travel(Someone, channel-tunnel) :- carry(eurostar, Someone)可以推导出新的命题(assertions)。

XML 可以用一个 Horn 逻辑解释器实现如下的推理查询:

```
                Someone=fred
travel(fred,Where)  => carry(eurostar,Someone)  =>true
                Where=channel-tunnel        Where=channel-tunnel
```

转换成 XML 表示如下:

```
Rule:
<hn>
  <relationship>
    <relator>travel</relator>
    <var>someone</var>
    <ind>channel-tunnel</ind>
  </relationship>
```

```
    <relationship>
      <relator>carry</relator>
      <ind>eurostar</ind>
      <var>someone</var>
    </relationship>
</hn>
Fact:
<hn>
    <relationship>
        <relator>carry</relator>
        <ind>eurostar</ind>
        <ind>fred</ind>
    </relationship>
</hn>
```

推理条件为:

```
<relationship>
    <relator>travel</relator>
    <ind>fred</ind>
    <var>where</var>
</relationship>
<relationship someone="fred" where="channel-tunnel">
    <relator>carry</relator>
    <ind>eurostar</ind>
    <var>someone</var>
</relationship>
```

推理结论为:

```
<ind where="channel-tunnel">
   true
</ind>
```

以上推理可以通过 SLD 解析实现,可以用 XML-QL 实现查询操作。

第6章　XML 与分布式计算

XML 以一种简单而直接的方式为处理数据开辟了一条新的道路。XML 文档结构化的固有属性已成为大多数消息系统的基础，缓解了区分数据和以不同的方式处理不同类型数据的需求。XML 在应用程序中作为一种在异构应用间传递数据的方法，提供内容之上的元信息，维护数据的结构。XML 将成为未来消息系统的一个主要技术。

6.1　分布式计算

什么是分布式计算？简单地说，分布式计算是两个或多个软件共享信息。这些软件既可以在同一台计算机上运行，也可在通过网络连起来的几台不同机器上运行。绝大多数的分布式计算是基于客户机/服务器模型的。在客户机/服务器模型内，有两类主要软件：客户机软件，它提出信息或服务的请求；服务器软件，提供这种信息或服务。

6.1.1　分布式计算的优点

分布式计算的主要优点是通过使用如下的技术更有效地使用计算资源：

(1) 稀有资源的共享。如果是一台高性能打印机或绘图仪，分布式计算使网络上的每个人都能使用它们，而不仅是那些在连接着打印机或绘图仪的机器上有账号的用户。

(2) 在许多不同机器上平衡计算负载。

(3) 把应用程序放在最符合需要的机器上。

共享稀有资源和平衡负载是计算机上行分布化和计算机下行分布化的核心思想之一，这两种技术都是把非分布式计算转向分布式计算。计算机上行分布化(Computer Upsizing)是指把业务从 PC 机上的非分布式计算转移到网络分布式计算(包括 PC 机、高性能服务器和工作站)的过程。计算机下行分布化(Computer Downsizing)是指把业务从实验室里的大型机上的非分布式计算转移到办公室里的工作站和 PC 机的分布式计算的过程。

6.1.2　分布式计算的现有机制

除了通用对象请求代理体系结构——CORBA 方案外，还有三种常用的方法来构造分布式方案：

(1) 分布式通用对象模型(DCOM)。

(2) 远程过程调用机制(RPC)。

(3) 网络应用程序编程接口(API)编码。

但是使用上述方案降低了开发业务问题解决方案这一主要任务的重要性，也就是说，当使用 RPC 或网络 API 时，除了解决业务问题外，必须另外处理基础设施的问题。

分布式计算倾向于集中用 RPC 机制在网络中对应用程序进行分布式处理。正如它的名字所隐含的那样，RPC 处理过程的调用。通常调用过程在本质上是同步的，这就意味着提出请求的应用程序在服务器完成该过程之前必须等待。使用 RPC 的应用程序常常必须理解网

络传输的一些概念,并负责定位满足请求的服务器。另外,RPC 通常要求在请求初始化的时候服务器必须已经启动,这就限制了能够完成请求的位置。DCE RPC 允许多个服务器的声明,但应用程序的编程人员必须自己发现这些服务器并选择一个。DCE RPC 没有一个代理来自动发现和选定服务器。

虽然网络 API 常常支持同步通信和异步通信,但它没有提供编码数据设备以使多个平台上有不同数据表示的应用程序能够理解这些数据。

使用网络 API 通常要求创建和使用额外的软件,这些软件通常是非常底层的软件,诸如处理不同平台上数据表示差异的软件。要创建这样的软件,就可能要回答诸如如何把 PC 机上的浮点数转换成 VAX 系统上的浮点数等问题。用户还要为打包和传送请求以及解包和把请求传递到例程而编写代码,可能还要做大量的缓冲区操作并注意不同平台的细小差别。

6.1.3 CORBA 如何增强分布式计算

CORBA 用如下手段增强分布式(客户机/服务器)计算:

(1) 支持客户机与服务器间灵活变化的关系;

(2) 加入一个称为代理的中介;

(3) 允许服务器有多个进程;

(4) 支持同步及异步两种通信形式。

在 CORBA 中,分布式计算和对象模型的结合实现了相互促进。CORBA 在分布式计算和对象模型环境中加入了如下内容:

(1) 分布式计算方面的增强。

对于分布式计算环境,CORBA 加入了特定对象的引用。在 CORBA 中,要完成某个操作,所需要做的仅是请求某个有能力完成该操作的对象去完成它。例如,如果已经有了一个账户对象的引用,就可以请求存款和取款操作。客户机的开发者不需要知道更多的信息。

(2) 对象模型方面的增强。

对于对象模型,CORBA 加入了代理的概念。代理使应用程序不需要知道对方在网络上哪个地方和对方是如何工作的就可以进行交互。客户机向代理发送一个请求,要求在一个对象上执行某个操作。只有代理需要知道 CORBA 服务器和客户机在网络上的位置。

CORBA 对象模型的加入提供了一个极好的方式来抽象客户机/服务器环境的一些固有复杂性。面向对象计算补充了分布式计算的优点,因为它隐藏了对象代理的功能,如发现服务器、定位服务器以及决定如何获得服务器的工作。另外,CORBA 对象请求代理还解决了不同程序语言间(如 C++,Smalltalk)的调用问题,不同对象模型间的调用问题,甚至是面向对象系统与非面向对象系统间的调用问题。

6.1.4 基于主体的分布计算模型

近年来,分别以 Internet 技术为核心的和以分布式对象技术为核心的应用系统集成对分布式客户机/服务器计算机系统集成平台提出了新的需求。集成平台不仅支持网络级和数据库级的集成,而且要支持各类应用程序的集成;能适应基于局域网的部门应用,而且要支持基于广域网的全球应用;不仅支持客户机独立地共享服务器资源,而且支持各客户之间进行协同工作。

为此提出基于主体的软件工程。这种软件开发方法的优点是便于软件的创建,有很强的

互操作能力。它代表了一种新的、功能强大的解决大规模软件工程问题的方法。在该方法中，应用程序编写为软件主体，即软件构件，这些构件之间通过主体通信语言可以进行比普通消息传递更规范、更明确的通信。主体通信语言的显著特征是其表达性，在异构环境下更能体现出面向主体编程方法的优越性。

基于主体的分布计算模型是由一组相互作用、协调工作的主体组成的计算系统，主体之间的交互关系为请求/服务关系，是一种完全对称的分布式系统。图 6.1 给出了基于主体的分布计算模型。在基于主体的分布计算模型中，系统基本单元的主体可以具有客户和服务器的功能，主体之间通过请求/服务方式，在约定的协作策略指导下协同工作。

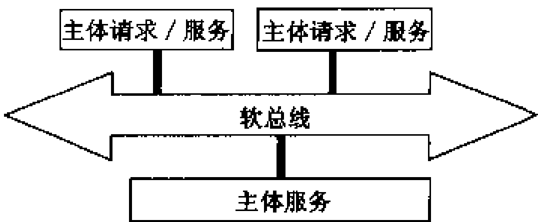


图 6.1 基于主体的分布计算模型

6.2 XML 与分布式计算

实现一个分布计算系统将面临三大挑战：数据传递、接口管理和远程调用。XML 的出现，对所有这些方面都提供了帮助，基于 XML 的语义消息有助于统一异构的分布式体系结构。

6.2.1 数据传递

多数通用的分布式计算模型，比如 DCE、DCOM、RMI 和 CORBA，都尝试着向开发者提供标准函数/方法调用范例，它和本地调用是完全相同的。如：

FOO(int a, char b, double c)

这通常是通过使用某一类型的接口定义语言(IDL)描述接口来达到的。IDL 是与操作系统、硬件和编程语言无关的，它为接口定义提供了一个标准的方法。对不同的操作系统/编程语言都存在着 IDL 编译器，因此允许通过操作系统和编程语言远程调用。由 IDL 编译器生成的编码(和下层的中间件服务)负责以发送方使用的格式从发送方来接收数据，并以接收者能理解的格式将数据呈现给接收方。

这一范例对开发者来说很方便，因为它隐藏了分布的方面，使其几乎是透明的。遗憾的是，这一方便带来的问题严重存在于通信应用之间。通常它不会对本地通信(应用程序内部)构成问题，但对企业内部的多个应用程序是一个真正的挑战，在企业间的通信甚至更难。

使用这一方法最典型的问题是：

(1) 数据格式：发送和接收应用程序必须具有相同的数据格式和顺序。

(2) 数据语义：接收方必须和发送方具有对参数语义的相同的理解(以顺序为基础)。

(3) 外部数据：由发送方提交的信息必须和接收方希望的一样。

几年来，语义消息为解决这些问题提供了方法。语义消息被以许多种不同的方法定义，一般来说，定义语义消息为纯的数据语义——包含数据和这一数据元素所代表含义的定义的消息。因此，处理语义消息的应用不是由数据顺序或其类型所驱动的，而是由命名约定(数据语义)驱动的。这一模式明显更适合于实现数据传输。

语义消息最简单的情况是名字/值对，它被分布计算模型内在地使用。传递的数据不是由其在数据流中的位置来定义，而是由这一特定数据块的名字来定义的，在这个意义上，这类消

息是自描述的。自描述的数据允许两个应用共享数据语义（名字），而不是在消息中一个内部数据表示和数据的顺序上达成一致。

自描述数据方法将所有的输入数据作为一个单独的消息提供给应用，而不是解析输入的信息并提供对每一条信息的访问（如基于 IDL 方法）。这种方法强迫一个应用通过解析输入的消息来抽取需要的信息。

XML 以一种统一的方式实现了任意复杂度的自描述结构化数据。标准化 XML 解析器和 XML 文档对象模型（DOM）的标准化表示极大地简化了 XML 数据的解析和抽取。基于 XML 的语义消息的一个典型例子如下：

```
<PurchaseOrder>
    <OrderHeader>
    <OrderID>5</OrderID>
    <BuyerID>111-0798</BuyerID>
    </OrderHeader>
</PurchaseOrder>
```

通过允许文档内的任意结构用法，XML 引入了任意的、复杂结构的数据语法。XML 另一个优点是它允许无类型数据的传输，并从等式中获取变量类型，因此每一个应用均可以将数据转换成其希望处理的类型。

分布系统的另一个问题是两个应用能共享相同的数据，但使用其不同的语义（比如，POID 与 PurchaseOrderNumber）。标准的 XML 工具比如 XSL 处理器，可以自动进行数据语义的转换，用不同的词汇简化系统的集成。

因为如下原因，利用语义消息能够建立耦合程度小得多的系统：

（1）数据元素在消息中的位置是无关的。发送方和接收方只需在数据语义（命名）上达成一致。

（2）接收应用只寻找预定义语义的数据。发送应用可自由地包含附加的数据而不会影响接收方。

（3）XML 文档与类型是否有效无关。发送方和接收方不需要在交换信息的数据类型上达成一致。

（4）接收方顺序地处理输入的语义消息。可为参数建立一系列的默认值，而不用提交给发送方。

标准的 XML DOM API 允许发送方和接收方建立和解析 XML 文档而不用考虑编程语言和操作系统。虽然基于 XML 的语义消息提供了一个有力的分布系统模型，但当实现这一方法时两件事情必须牢记：

（1）语义消息的使用为分布系统的创建增加了编程工作。在这种方法中，处理输入数据是应用程序员的责任，而不是 IDL 编译生成的编码的责任。

（2）语义消息的处理编码增加了接收语义消息的组件实际执行的延迟；但由于 XML 语义消息通常以字符串的形式传输，因而潜在的分布系统软件不需要打包和解析数据，因此这一延迟部分地得到了补偿。

XML 有效性（DTD 或模式）是否应当应用于数据传输？这种情况下使用 XML 有效性将有悖于语义消息的目的，因为其目的是使系统间的耦合不强。引入 XML 有效性与使用 IDL 相似，如果这样做了，语义接口的灵活性就消失了。因此使用语义消息能够创建灵活、松耦合的系统。

6.2.2 接口管理

当系统增长时，消息管理成为一个日渐突出的问题。这里的难题不仅是新的功能，而且所需的每个输入的参数组合都需要创建一个新接口。比如，如果 x 的值能以两种方式计算，以参数 a 和 b 或者 a 和 c 为基础，则需要两个接口：

```
Foo1(double x, int a, char b)
Foo2(double x, int a, double c)
```

因为这两个接口本质上都支持相同的功能，这就导致要创建给出所有可行参数的很"宽"的接口，对上面的例子而言就是：

```
Foo(double x, int a, char b, double c)
```

这种方法减少了所需接口的数目，但强迫所有引用它的应用提交所有参数包括它们没有的参数。为解决这个问题，引入一个标志系统，或者一个相似的东西来表明某些参数不存在。继续上面的例子，我们将得到类似于如下的接口：

```
Foo(double x, in t a, char b, double c, Boolean type)
```

这里"type"的值定义调用的实际值。

所有的方法在引入数据语义的专有实现上都是相等的。由于这一解决方案的专有属性，它不能被一般化，因此为每一个接口都需要一个新的实现。

基于 XML 的语义消息允许为所有需要的函数创建一个单独的接口，有一个定义明确的特征即使用 XML 文档形式的输入和输出参数。比如说：

```
Foo(xmldoc input, xmldoc output)
```

输入文档包含了输入参数的任何允许的结合，输出文档包含了相应于接收到的输入参数的输出的结合，对上面的例子而言，输入文档可以是下列之一：

```
<input>
    <a>5</a>
    <b>abc</b>
</input>
```

或

```
<input>
    <a>5</a>
    <c>78</c>
</input>
```

或者甚至是(如果 b 或 c 的正确的默认值存在)：

```
<input>
    <a>5</a>
</input>
```

在上面例子里，相同的输入文档包含发送方为实际的调用提交的不同类型的数据。因为输入文档包含信息语义(数据元素的名字)，故不需要任何附加的"标志"来定义实际提交哪一

个信息。接收应用能通过浏览输入文档获得这一信息。

这一方法同样简化了接口的维护。在标准的分布式环境下，每次系统的一个接口改变时，相应的变化必须应用于这个接口的所有用户。更有甚者，所有这些变化必须同步实现，否则整个系统将停止功能。语义消息，它提供了一个更松散的耦合系统，通常通过在应用自身中程序化地处理参数而简化这一问题。正如前面所讨论的，一个默认值定义明确的系统能允许同时在发送和接收应用中不断增加变化。

6.2.3 远程调用

所有的分布环境都以同样的方法实现远程调用：在发送方的地址空间里创建一个远程组件(对象)的代理，在接收方的地址空间里创建一个 Stub。发送方与代理通信，其依次与和接收方通信的 Stub 交谈(如图 6.2)。

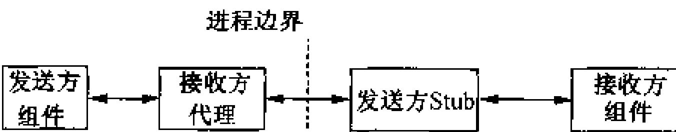


图 6.2 分布式体系结构的典型实现

代理和 Stub 的引入，虽然简化了编程模型，但使整个系统变得复杂，导致更昂贵的计算和存储费用。在这种体系结构下，组件间的远程通信封装在代理/Stub 通信中，它是由建立在中间件 API 基础上的 IDL 编译器生成的。发送方只和本地代理通信，接收方从本地 Stub 得到所有的请求。

因为代理和 Stub 在发送方和接收方的存储空间都要被创建，故每个连接需要另外的存储器，在图 6.3 所示的简单例子中，来自客户方的一个组件要处理服务器进程中与四个不同组件的连接。客户进程包含 4 个代理和 4 个服务器 Stub。若服务器支持 100 个客户，则存在 400 个 Stub，其总计达到的存储需求相当大。这些 Stub 同样必须在同一点创建和消除，增加了全局的计算费用。

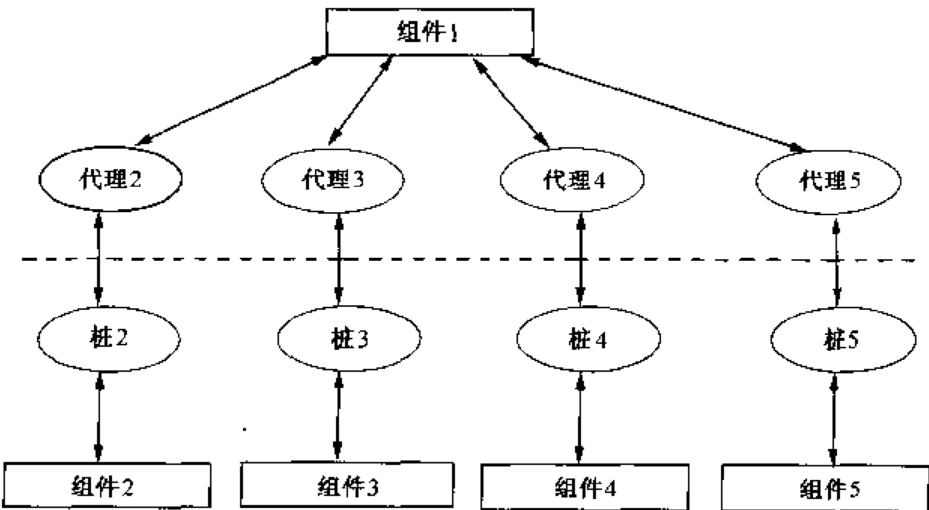


图 6.3 不同进程的组件连接

在代理/Stub对之间需要创建一个单独的逻辑连接。虽然大多数的分布模型实现能共享底层的物理连接(通常是TCP/IP),但建立一个代理/Stub连接是一个昂贵的操作。因此一般的结果是所有的分布计算模型推荐在发送方一次建立连接,在组件的生命周期中一直保持其开放。这就导致接收方和发送方组件的生命周期必须同步。当接收方是一个服务器,这会导致可扩展性的减弱(由于存储器的使用),这通常是不能接受的。DCOM和EJB规范都引入中介的上下文对象来解决这一问题。

基于XML的语义消息对无状态组件提供了一个很完美的解决方案(如图6.4)。这里的无状态指的是接收方没有传统的状态。无状态组件仍能具有持久的状态,保存在数据库中。

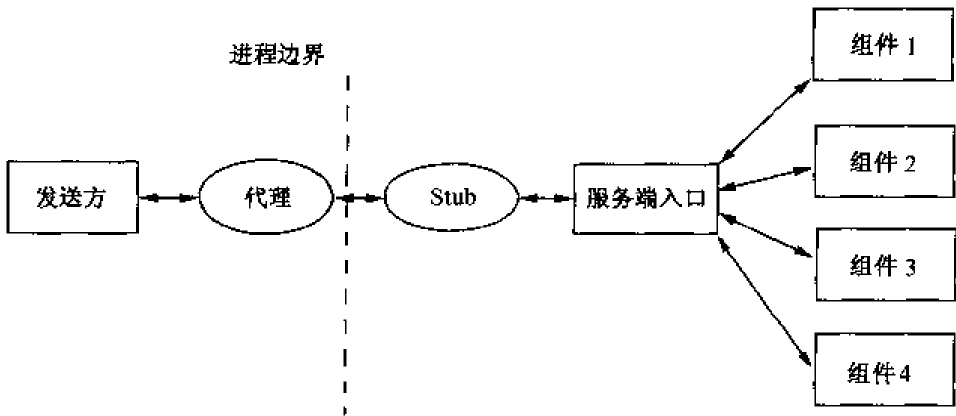


图6.4　使用服务端入口组件来最小化代理/Stub对的数目

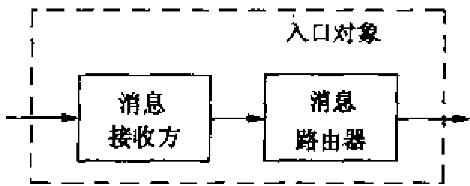


图6.5　入口对象剖析

一个"服务端入口(Gate)"对象被引入来代表进程中所有的组件。它能发送消息给进程而不是进程的任何特定组件。在这种情况下,代理/Stub对的数目是1(对服务端入口对象而言),不像图6.2中是4。在进程中一个单独的服务端入口接收指定给所有组件的所有消息并妥善安排实际的执行。因为XML不在消息语义上强加任何限制,故可能在标准的封装上放置每一个消息。这一封装包括消息要发送到的组件的组件名和方法名。上述的服务端入口对象包括两个主要的部分(如图6.5)。

消息接收方是特定协议(DCOM,CORBA,RMI等)上的一个监听者。这一对象负责接收使用预定义的通信协议的消息。在接收到消息以后,剩下的通信端部分对接收进程而言是内部的。消息路由器负责解析封装并抽取组件的名字,消息设定的方法以及消息本身。然后路由器实例化相应的组件,将控制传递给它来执行,并在进程完成时删除它。

服务端入口对象的思想并不是一个新思想。它和"facade"方式或者由J2EE引入的无状态会话beans有某些相似。这里最大的不同是使用了基于XML语义消息的能力和灵活性,其标准化了服务端入口接口并使服务端入口和支持组件间的关系成为动态的。这是惟一不需要重写服务端入口来支持附加的组件或组件上附加的方法的一种体系结构。在现代大多数中间件环境中引入对内省Introspecition的支持,使得路由器的实现更为简单。

到现在为止,我们讨论了使用服务端入口对象来在进程中对组件进行多路访问。相同的体系结构同样能用于对基于某一中间件产品的分布式系统的多路访问。在这种情况下,服务端入口能司职"入口"和"桥",意味着除了多路复用,它能在多个中间件类型之间转换消息。比

如说，当内部的通信可以是 DCOM 时，服务端入口能接收 RMI 消息。

下面分析运行在不同中间件产品之上的“桥接”组件的复杂性，研究基于 XML 的语义消息如何帮助解决这些复杂性。

名字分析

不同的中间件系统通常具有不同的名字服务，这使得名字分析在本质上就很复杂。对将要调用运行在不同中间件上的每一个组件都必须完成这一名字分析。通过增强图 6.4 所示的体系结构，将名字分析功能移到对外的服务端入口是可能的，入口减少了每一个组件的名字分析操作并封装了这个操作。基本上，如果没有在本地中间件平台内部发现一个组件，请求就被传送到对外的服务端入口，它必须分析名字并传递请求到接收方对内的服务端入口。对内/对外的服务端入口的推荐系统使得名字分析的粒度大大降低，简化了这一问题。

组件调用

不同的中间件产品具有不同的代理/Stub 实现模式，大多数的桥接方案使用两个代理(本地代理和来自接收方中间件的代理)来进行远程调用。引入一般的服务端入口体系结构(见图 6.6)通过限制调用需求简化了这一机制，只有对外的服务端入口必须调用相应的对内服务端入口。

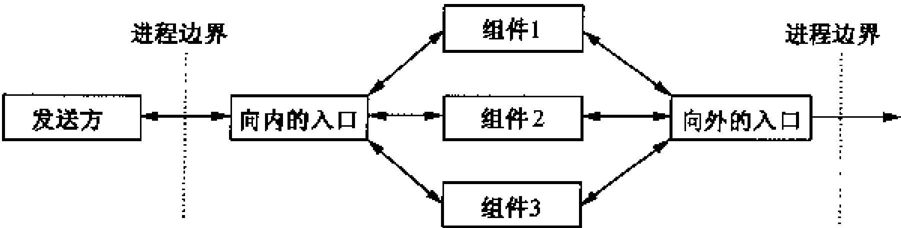


图 6.6 服务端入口的一般化

数据转换

每一个中间件产品需实现其自己的数据打包、表示和转换途径。其结果是多个中间件包之间同步数据的表示常成为一个挑战。基于 XML 的消息通过在多个中间件系统之间使用数据的字符串表示极大地简化了这一问题，这一表示是非常标准的。

基于服务端入口的体系结构解决了如下的问题：

(1) 将需要的代理/Stub 组合数目减少为每个进程一个与每个组件一个。

(2) 不需要耦合组件的生命周期来提高性能。

(3) 由于是以进程为基础而不是以组件为基础，载入平衡在这种体系结构中被简化。

(4) 多个中间件平台桥接成一个无缝的环境。

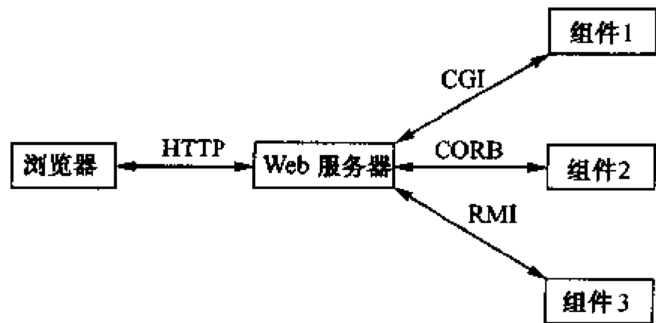


图 6.7 Web 服务器作为一个入口

在以服务端入口为基础的方法中为简化消息路由器的实现，XML 有效性在处理封装的层次上是有意义的，但我们仍假设消息本身是未经检验的 XML。

这种类型的体系结构是当今 Web 技术的基础。Web 服务器可被看做一个桥接 HTTP 通信和系统内部中间件的一个复杂的服务端入口，不管中间件是 CGI、RMI、CORBA 或者 DCOM(见图 6.7)。

传统的路由是以提交给 Web 服务器的语义/值对上的 URL 和数据为基础的。B2B 和 XML 的发展通过引入以 XML 为基础的语义消息和合并路由的信息和数据，正改变着这一途径（和前面提及的解决方案非常相似）。在 Web 上实现 XML 语义消息的例子是 XML 和 B2B 服务器和 Web 服务。

6.2.4 统一的分布式系统体系结构

使用基于 XML 的语义消息和基于服务端入口的体系结构来统一 Web 和非 Web 情况，就能创建一个通用的体系结构，并提供任何类型的访问协议（HTTP、CORBA、DCOM）（见图 6.8）。

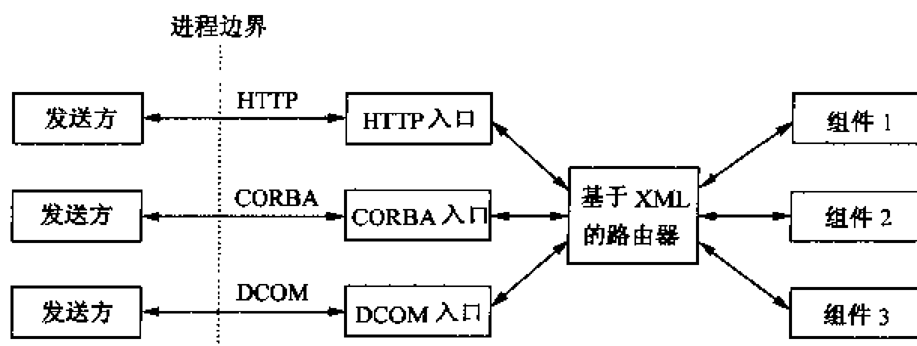


图 6.8 通用分布式系统体系结构

该体系结构以一个标准的方式桥接运行在不同中间件平台上的多个系统。这一通用体系的结构优点是：

(1) 基于 XML 的语义消息允许在多个系统间的灵活、松耦合的消息传送。

(2) 服务端入口对象与基于 XML 的语义消息相结合简化了复杂的操作，比如名字分析、远程调用和异构中间件环境间的数据传输。

(3) 易于扩展，因为在现有组件上引入另外的组件和（或）方法不需要对控制组件访问的路由器进行修改。

对该体系结构的扩展是会话状态支持，这能很容易地引入类似于 Web 服务器支持的会话。组件的会话状态能具体化为 XML 文档的形式并存入 XML 路由器中。当一个新的请求到达时，路由器能将状态信息附加到请求数据后。

在实现这一体系结构时，正确使用基于 XML 的语义消息是很重要的。虽然使用语义消息给整个系统增加了适应性和灵活性，它同样增加整个执行的延迟。因此最好采用如下方式使用基于 XML 的语义消息：

(1) 用于多个系统和（或）进程间的通信。

(2) 在一特定的进程中作为一个通信的内部中间件平台。

6.3 集成 XML 和 CORBA

XML 和 CORBA 是建立分布式系统的关键技术，但它们都是独立发展的，涉及到内容管理和分布式应用的不同需求。XML 的目的是对文本标记人可读的文档的存储和操纵，而中间件解决方案如 CORBA，目的是将协作的计算机应用程序联系在一起，交换临时的数据。这些

数据可能从不为人所直接阅读。XML 和中间件是互相补充的技术，这两种技术决不能互相替代。实际上，它们可以彼此互相借鉴发展，更会越来越多地结合在一起使用。

6.3.1 中间件技术

中间件技术是分布式系统中的关键技术。中间件是处于应用程序及应用程序所在系统之间的软件。中间件把应用程序与系统所依附软件的较底层细节和复杂性隔离开来，使应用程序开发者只处理某种类型的单个 API——而其他细节则由中间件处理。应用程序开发人员可以使用中间件而工作在应用程序较高的层次上，中间件提供了底低层的细节，这样开发人员就不用为操作系统和低层接口编码。为了解决用户的异构和分布问题，厂商们提供了一些拥有标准编程接口和协议的分布式系统服务，这些服务就称为中间件服务。中间件服务是介于平台和应用间的通用服务。这里的平台指的是处理器架构和操作系统应用编程接口(API)所定义的一系列底层服务和处理元素。一项中间件服务是由其支持的 API 函数和协议所定义的，因此对应于不同的接口和协议它存在多种实现方式。例如 DCE RPC 在不同的平台上都有其实现版本。中间件服务的定义是模糊的，这体现在中间件和平台的关系上。随着时间的推移，中间件和平台间可能互相转化，互相渗透。现有平台中的某个工具以后可能会变成中间件，这样做的目的可能是为了简化 OS 实现或让这一服务在所有平台上通用。

中间件服务也存在着不足之处：

(1) 理论与实践存在差距。许多流行的中间件服务采用专用 API 和专用或非公开协议，使互操作性和应用移植性受到影响，有碍实现。

(2) 中间件服务的数量和质量严重不足。当不得不考虑每一服务的全套 API 时(包括服务请求、系统管理接口和数据定义工具)，一些中间件服务甚至会带来大量的计算复杂性。为了简化计算环境的管理，开发者只能选择一小部分中间件来满足他们对功能和平台覆盖度的要求。

(3) 提高分布式应用编程的抽象度后，应用开发者仍面临困难的设计选择。开发者还得决定将这些功能放在分布式应用的客户端还是服务器端。一般来说，将表示层服务放在客户端，接近显示部分，而将数据服务放在服务器端，接近数据库。但这并非固定不变，对于任何具体情况都需要具体分析。

6.3.2 集成 XML 和 CORBA

6.3.2.1 XML 和 CORBA

CORBA 是由对象管理组织(OMG)建立的规范。对象是使用接口定义语言(IDL)来描述的，分布在通信总线即对象请求代理(ORB)上。这些对象用一种编程语言实现，典型的是 Java 或 C++，ORB 透明地处理所有网络、平台和语言问题。另外，使用 CORBA 服务，对象可被做成事务的、安全的等。其他销售商提供的工具帮助操纵和管理这些对象，实现大规模分布系统的开发。

一个分布式应用包含了三个逻辑组件：数据、逻辑和传输。CORBA 将这三个组件以分布式对象的形式联系在一起，其中使用一些面向对象的语言来实现逻辑。数据与传输相关，在客户方和服务器之间以二进制形式传输。相似的功能调用方式也用于在远程对象上透明地调用方法。相反 XML 只是关于数据，它以文本形式表示，并不天然地与任何传输协议相关联。它

可被任何能传输文本数据的传输协议所携带。XML 编程模型包含遍历一解析后的 XML 文档并在所遇到的元素基础上采取相应行为。对数据使用 XML 就能描述任意复杂的数据类型,建立松耦合系统,很容易地将一种数据转换成另一种形式,很容易地在浏览器上显示数据。XML 解析器对大多数语言和操作系统是可行的,因此可以容易地将解析器嵌入到一个应用中。流行的浏览器都支持 XML,这使开发者能显示 XML 数据。CORBA 已被成功地用于与 legacy 和 ERP 系统接口。使用 XML 而不用 IDL 类型可能在某些情况下让这一集成更加容易。很多 ERP 销售商已经保证其产品都支持 XML,这将使 XML 的使用更具吸引力。

一个分布式系统使用 CORBA 来实现分布,使用 XML 管理数据就会充分利用两种技术的的优点,并且同时建立在两个开放工业标准上。

6.3.2.2 XML 和 CORBA 集成的三种方法

如表 6.1 如示,XML 和 CORBA 在概念上有一些共同之处。

表 6.1 XML 和 CORBA:共同概念

概　念	XML	CORBA
数据	XML 文档	CORBA 简单类型、结构、顺序、Any、DynAny、值类型
类型规范	XML DTD,XML 模式	CORBA 类型码
名字空间	XML 名字空间	CORBA 名字空间

XML 文档是天然的文本,它定义了两个通用的编程 API:

(1) 由 W3C 定义的文档对象模型(DOM)。该 API 提供对独立于文档词汇的 XML 文档的访问,也就是说 API 方法是一般的文档处理方法。它从一个 XML 文档创建一个全文档对象,因此如果处理大文档,就需要足够的空间。

(2) XML 的简单 API(SAX)。这是一个事件驱动的 API,只有感兴趣的文档才被加载并可使用。

这两种 API 都是很普遍的,另一个可选的方法是提供一个 API,其中 XML 文档的词汇直接映射到了 API。比如,直接使用出现在文档中的元素和属性名来访问文档。在这种方法中,XML 文档的语义对操纵它的编码来说是可见的,因而更易于理解。

CORBA 对象的接口和其可处理的数据类型使用 IDL 来定义。因此 XML 与 CORBA 的集成必须使用 IDL 来表达。但是当与 IIOP 兼容的结构能从 XML 文档直接创建时,有可能动态地执行这一转换。

IDL 提供了以 CORBA Any 形式的结构化和聚集数据类型,同时也提供标准的简单数据类型。

使用 CORBA 对象类型产生一个 XML IDL API 并不值得。这将意味着 XML 文档封装在一个 CORBA 接口中,文档以一远程方式访问,因此这是不够的。

人们更希望使用值类型或者现存的 IDL 数据类型作为一种传值到文档的方式,因为这将允许在 XML 文档上进行有效的操作。值类型规范只是最近才被 OMG 批准,而且这一规范很少有实现方式可用。因此关于 XML 和 CORBA 的集成是使用值类型还是直接映射到现存的 CORBA 数据类型仍然存在一个选择。

方法一:简单的方法

最初的方法也是最简单的方法,传送一个 XML 文档到一个 CORBA 对象,该 XML 文档

被“字符串化”。

```
Struct XMLDocument
{
  String dtd;
  String document;
};
typedef sequence<XMLDocument> XMLDocuments;
interface MyObject
{
  void invoke(XMLDocuments documents);
};
```

这一方法的优点是它易于扩展现存的对象接口来支持 XML 文档。这一方法的缺点也是明显的。

(1) 空间的低效使用。在一个分布的应用中,冗长的 XML 文档分布在网络上。

(2) 时间的低效使用。XML 文档必须在使用的每一个地方进行解析。

(3) 不是类型安全的。没有办法在传输时确认 XML 文档,而它必须在使用的每一个地方被确认。

方法 2:使用值类型映射 XML 到 IDL

这一方法使用 CORBA 值类型,而它正在由 OMG 讨论。CORBA 值类型的使用已提供了尽可能多的 DOM API。值类型作为一种新的 IDL 模型,是现存的结构和接口的交叉。它常常通过引用来传递,但可包含方法说明,比如一个接口。

因为 XML 文档的文档、元素和文本部分实现了 Node 接口,故 Node 类型是 DOM API 的中心。API 使用一个标准的命名协定来访问 XML 文档的所有部分。

其他重要的 DOM 类型是 DOMString、NodeList、Document、Element、Attr 和 Text。这一方法和标准的 DOM API 一致。但是,XML 文档的语义在使用一个标准的 API 时并不清楚。因为值类型规范的 CORBA 实现方式还没有成为公认的方式,因此这一方法得以实现可能还有一段时间。

方法 3:映射 XML 到 CORBA IDL

这一方法利用现有的 CORBA 实现方式普遍支持的 IDL 类型。它在调用操作前映射一个 XML 文档到 CORBA 类型,并在接收对象接收之前将 CORBA 对象转换回 XML。这一方法对网络是很有效的,并且是类型安全的。另一个优点——当被直接转换成用于封装文档的 CORBA 结构时,文档的语义仍被保留。

下面的例子将演示一个 XML 文档和其相关的 CORBA IDL 定义之间的映射。当给出相应 XML 文档的 DTD(或 XML 模式)时,这一映射可自动地执行。

XML 文档实现为一个层次的 CORBA 类型,使用一个 XML 和 IDL 及 CORBA Any 间的双程映射。在这种情况下,CORBA Any 将始终维持一个 CORBA 结构,这一结构的内容依赖于 XML 文档。

(1) 映射 XML 到 IDL:

① 一个 XML 元素映射到一个 CORBA 结构,一个相应的简单类型用于元素的值。

```
<element>
```

```
  1.23
</element>

struct element
{
  float -Value;
}
```

② 一个 XML 属性映射到 CORBA 结构的字符串成员。

```
<element color='green' font='fixed'/>
struct elment
{
  string color,
  string font
}

<element color='green'>
1.23
</element>
struct element
{
  string color,
  float-Value;
}
```

③ XML 元素的单个子女映射到父结构内一个结构。

```
<element>
  <child id='abc'/>
</element>

struct element
{
  struct child
{
  string id;
}
}
```

④ 具有相同名字的数目不定的 XML 子元素映射到一结构内的一系列结构。

```
<elements>
  <element color='green'/>
  <element color='blue'/>
</elements>
```

```
struct elements
{
  struct element
  {
     string color;
  }
  sequence<element> -element;
}
```

⑤ 具有相同名字、数目固定的子元素映射到一结构内一个结构数组。

```
<elements>
  <element color='green'/>
  <element color='blue'/>
</elements>
struct element
{
  string color;
}
struct elements
{
  element -element[2];
}
```

⑥ 产生一个 CORBA 类型码来描述结果 CORBA Any 的新定义的结构和窗体部分。

⑦ 一个使用 XML 文档的对象可用 IDL 定义如下。

```
interface MyObject
{
  void invoke(Any document);
}
```

(2) 映射 IDL 到 XML:

① 只有结构、顺序、数组和基本类型被映射。

② 一个 XML 模式或者 DTD 从 CORBA 类型码中产生。

③ 一个结构成员映射到除"-Value"外的所有属性,"-Value"映射到元素的文本值。

④ 序列和数组映射到多个元素。

6.4 XML-RPC

XML-RPC 是 Internet 上执行的远程过程调用,它执行存在于网络或 Internet 上的分布式进程。

6.4.1 RPC

RPC 即远程过程调用,是由 UNIX 系统引入的,但 RPC 编码在不同的 UNIX 操作系统间

并不是可移植的。RPC 的目的是允许程序员开发网络应用而不需要他们成为 Socket 程序员。RPC 提供一个接口定义语言从而能为多种编程语言生成通信 Stubs 和骨架。这是通过使用特定 IDL 之上的一个编译器程序来实现的。

后来，专门成立一个协会开发称为 DCE RPC 的 RPC 机制，这一机制能运行在不同操作系统间。它提供一个编译器和库，其 Stub 能在多种操作系统上被编译和执行，允许客户方和服务器编写可在异构操作系统间移植的编码。开发者只需要在新的平台上编译编码，应用程序就可工作了。但 DCE 本身并不是设计来处理面向对象语言的。

OMG(对象管理组织)定义了对象请求代理(ORB)规范即 CORBA 来为 RPC 提供一个面向对象的机制。CORBA 和 RPC 相似，提供一个由开发者使用的 IDL 来为多种语言定义对象接口。但是，由 CORBA IDL 编译器开发的 Stub 和骨架并不必在多个 ORB 销售商间移植。通过使用通信协议 IIOP(Internet Inter-Orb Protocol)，异构的 CORBA 服务器间的通信被标准化了。

Java 平台提供了它自己的面向对象的 RPC 机制，称为远程方法调用(RMI)。它提供了一个本地的 Java 机制以允许对象在内存中多个虚拟机器间通信。这一通信允许本地对象调用方法和远程对象，就跟它们位于同一进程空间一样。RMI 使用 RMI URL 来定位一个在另一虚拟机的远程对象。使用 Java 的对象串行化(Serialization)，一个 RMI 对象能被在虚拟机间传输及被分布式进程访问。RMI 的对象串行化是 RMI 和 CORBA 间主要的不同之一。另一个不同是 CORBA 提供了一个跨平台、跨语言的体系结构，而 RMI 是被设计来解决 Java 世界的分布式对象通信问题的。最后一个关于 RMI 的有意思的地方是它能在 HTTP 上建立隧道。

6.4.2 RPC 和 XML

那么，什么是 XML-RPC 呢？XML-RPC 是 Internet 上执行的远程过程调用。这就是说它能够升级 HTTP 协议以执行存在于网络或 Internet 上的分布式进程。XML-PRC 被一个具有特殊目的的 HTTP 服务器所调用。

XML-RPC 请求使用 HTTP POST 发送。一个 HTTP POST 允许一个客户在应用和 Web 服务器间发送大量的数据。使用 XML 来定义 HTTP 请求的体，对请求的响应也以 XML 返回。HTTP 请求的 XML 的第一部分包含一个对 POST 可选的 URI，一个必需的用户主体和需要识别的主机名，一个 text/xml 内容类型和对请求的内容长度。URI 帮助选择由请求到处理它的调用间的路由，如果服务器只处理 XML-RPC 请求，则它可以是空的。请求的第一部分的例子如下：

```
Post /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2(WinNT)
Host:rpcxml.experiments.com
Content-Type: text/xml
Content-length:199
```

在这个例子中，所有请求都路由到主机 rpcxml.experiments.com 上的 RPC2 应答器，运行一个前端服务器，内容类型是 text/xml，长度是 199 个字符。

XML-RPC HTTP 请求的第二部分是由单个 XML <methodCall> 结构组成的。在 <methodCall> 中一个 <methodName> 标签用于识别被调用的方法的名字。名字可用于表示一个脚本文件，负责回答请求，或者表示具有文档层次的文件。

参数在<methodCall>标签内部用<params>标签来表示。一个单数的<param>标签在<params>标签内使用，用于识别传递给过程的每一个参数。每一个<param>标签包含一个<value>标签。<value>标签包含参数类型和对其值的描述。请求第二部分的一个例子如下：

```
<? xml version="1.0">
<methodCall>
<methodName>experiments.getDNSName</methodName>
<params>
  <param>
    <value><string>129.107.10.3</string></value>
  </param>
</params>
</methodCall>
```

在这个例子中，方法名表示一个称为"experiments"的对象，它具有一个方法称为"getDNSName"。这个过程具有一个类型参数"string"和值"129.107.10.3"。而过程的返回类型也将是一个 XML 结构。

响应的格式分为一个 HTTP 响应头和 XML 结构。响应的体是一个单独的 XML 结构，即一个<methodResponse>，其包含一个惟一的<params>标签。<params>标签包含一个<param>标签，而<param>又包含一个<value>。<value>标签包含返回结果的类型和结果的值。XML 结构的例子如下：

```
<? xml version="1.0"? >
<methodResponse>
  <params>
    <param>
      <value><string>pokemon.experiment.com</string></value>
    </param>
  </params>
</methodResponse>
```

在这个例子中，方法响应包含一个字符串结果，其值为"pokenmon.experiment.com"。响应同样能返回一个错误结构和一个错误码，由一个错误码 ID 和一个错误描述字符串组成。

根据这一协议可以设计用来跨越防火墙的 HTTP 协议。防火墙能过滤带有 text/xml 内容类型的 HTTP POST 请求。协议要求 Web 服务器提供 CGI 支持。协议的编程对象是 HTML 开发人员。RMI 之上的 XML-RPC 的一个优点是客户方开发人员不需要写 Java 代码，只需创建要发送的 XML 结构和解析返回的 XML 结构。

6.4.3 XML-RPC 与 Java

人们认为 XML-RPC 和 Java 是互相补充的技术。Java 的 RMI 技术大大增强了分布计算的能力。一个 Java servlet 被创建来处理 XML-RPC POST 请求。定义在<methodCall>标签内的<methodName>包含了 RMI URI 并附加了被调用的方法名。这一信息放置在<methodName>标签之内。方法名的一个例子如下：

<methodName>rmi://rpcxml.exp.com/networkinfo/getDNSName</methodName>

在这个例子中，RMI URL 定义了在主机 rpcxml.exp.com 上一个远程对象调用信息。远程对象上执行的方法名字是 getDNSName。

Servlet 负责从<methodName>标签检索 RMI URL 并寻找 RMI 注册表中的对象。一旦对象被检索到，servlet 使用 Java 的映像(Reflection)来访问被调用的方法。方法的参数可从包含在 XML 结构中的<params>信息来构建，其执行使用映像 API 来进行。返回的对象被分解成 XML 结构并返回到客户。

Servlet 负责解析输入的 XML 结构并生成输出的 XML 结构。它同样负责映射 XML-RPC 数据类型到 Java 对象，以及反过来的映射。如表 6.2 所示。

表 6.2 XML-RPC 到 Java 映射

XML-RPC 数据类型	解析器生成的数据类型	调用方希望作为 RMI 对象的输入参数的类型
<i4>或<int>	Java.lang.integer	Int
<boolean>	Java.lang.Boolean	Boolean
<string>	Java.lang.String	Java.lang.String
<double>	Java.lang.Double	Double
<dateTime.iso8601>	Java.util.Date	Java.util.Date
<struct>	Java.util.Hashtable	Java.util.Hashtable
<array>	Java.util.Vector	Java.util.Vector
<base64>	Byte[]	Byte[]

如果 servlet 不能够解析 XML<param><value>，映射将会向客户应用返回一个错误。错误被 servlet 通过捕获 Java 异常和生成 XML RPC 错误结构来进行处理。这些结构需包含一个指派给异常类型的数字值和与异常相关联的文本描述。

XML-RPC 的一个目标就是提供对分布式资源的方便的访问，由 XML-RPC 协议提供的 Java 抽象支持这一概念。这些技术的结合允许非 Java 的客户应用 HTML 页面来访问分布式 RMI 对象，它允许 HTML 开发者使用 RMI 进程而不需要用 Java 编码。这一结合的另一优点是 Java 应用是可在操作系统间移植的，这使得它们特别适合开发分布式的对象应用。

RPC 从其出现以来已经经历了巨大的发展。随着公司扩展其业务到 Internet 空间，各种技术如 Servlet、RMI、CORBA 和 XML-RPCs 对于访问 Web 上的服务将越来越重要。

6.5 简单对象访问协议——SOAP

SOAP 是用在分散或分布的环境中交换信息的简单协议，它是一个基于 XML 的协议。SOAP 宣称是一个无所不在的 XML 分布式计算底层结构的规范，可被设计为与各种其他协议结合使用。

6.5.1 简介

SOAP 以 XML 形式提供了一个简单的用于在分散或分布环境中交换结构化和类型化信

息的机制。SOAP 本身并没有定义任何应用程序语义,如编程模型或特定语义的实现。实际上它通过提供一个有标准组件的包模型和在模块中编码数据的机制,定义了一个简单的表示应用程序语义的机制。这使 SOAP 能够被用于从消息传递到 RPC 的各种系统。

SOAP 包括三个部分:

(1) SOAP 封装结构定义了一个整体框架用来表示消息中包含什么内容,谁来处理这些内容以及这些内容是可选的还是必须的。

(2) SOAP 编码规则定义了用以交换应用程序定义的数据类型实例的一系列机制。

(3) SOAP RPC 表示定义了一个用来表示远程过程调用和应答的协定。

虽然这三个部分都作为 SOAP 的一部分被描述,但它们在功能上是相交的。封装和编码规则是在不同的名字空间中定义的,这种模块化的定义方法增加了简单性。

在 SOAP 封装,SOAP 编码规则和 SOAP RPC 协定之外,SOAP 规范还定义了两个协议的绑定,描述了在有或没有 HTTP 扩展框架的情况下,SOAP 消息如何包含在 HTTP 消息中被传送。

SOAP 的主要设计目标是简单性和可扩展性,这意味着传统的消息系统和分布对象系统的某些性质不是 SOAP 规范的一部分。这些性质包括:

(1) 分布式碎片收集。

(2) 成批传送消息。

(3) 对象引用(要求分布式碎片收集)。

(4) 激活机制(要求对象引用)。

下面是 SOAP 消息的一个例子,在这个例子中,GetLastTradePrice SOAP 请求被发往 StockQuote 服务。这个请求携带一个字符串参数和 ticker 符号,在 SOAP 应答中返回一个浮点数。XML 名字空间用来区分 SOAP 标志符和应用程序特定的标志符。这个例子说明了 HTTP 绑定。SOAP 中管理 XML 负载的规则完全独立于 HTTP 是没有意义的,因为事实上该负载是由 HTTP 携带的。

例 1　在 HTTP 请求中嵌入 SOAP 消息。

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction:
"Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

下面是一条应答消息，包括 HTTP 消息，SOAP 消息是其具体内容。

例 2　在 HTTP 应答中嵌入 SOAP 消息。

```
HTTP/1.1 200 OK
Content-Type: text/xml;
charset = "utf-8"
Content-Length:
nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV = "http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m = "Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.5.2　SOAP 消息交换模型

SOAP 消息从发送方到接收方是单向传送，但正如前述，SOAP 消息经常以请求/应答的方式实现。

SOAP 实现可以通过开发特定网络系统的特性来进行优化。例如：HTTP 绑定使 SOAP 应答消息以 HTTP 应答的方式传输，并使用同一个连接返回请求。

无论 SOAP 被绑定到哪个协议，SOAP 消息采用所谓的"消息路径"发送，这使在终节点之外的中间节点可以处理消息。

一个接收 SOAP 消息的 SOAP 应用程序必须按顺序执行以下的动作来处理消息：

(1) 识别应用程序想要的 SOAP 消息的所有部分。

(2) 检验应用程序是否支持第一步中识别的消息中所有必须部分并处理它。如果不支持，则丢弃消息。在不影响处理结果的情况下，处理器可能忽略第一步中识别出的可选部分。

(3) 如果这个 SOAP 应用程序不是这个消息的最终目的地，则在转发消息之前删除第一步中识别出来的所有部分。

为了正确处理一条消息或者消息的一部分，SOAP 处理器需要理解：所用的交换方式（单向，请求/应答，多路发送等），这种方式下接收者的任务，RPC 机制（如果有的话）的使用，数据的表现方法或编码，还有其他必须的语义。

尽管属性比如 SOAP encodingstyle 可以用于描述一个消息的某些方面，但这个规范并不强制所有的接收方也必须有同样的属性并取同样的属性值。举个例子，某一特定的应用可能知道一个元素表示一条遵循约定的 RPC 请求，但是另外一些应用可能认为指向该元素的所有消息都用单向传输，而不是请求应答模式。交互双方的 SOAP 消息并不一定要遵循同样的格式设定，而只需要以一种双方可理解的格式交换信息就可以了。

6.5.3 与 XML 的关系

所有的 SOAP 消息都使用 XML 形式编码。

一个 SOAP 应用程序产生的消息中，所有由 SOAP 定义的元素和属性中必须包括正确的名字空间。SOAP 应用程序必须能够处理它接收到的消息中的 SOAP 名字空间，并且它可以处理没有 SOAP 名字空间的 SOAP 消息，就像它们有正确的名字空间一样。

SOAP 定义了两个名字空间：

(1) SOAP 封装的名字空间标志符是"http://schemas.xmlsoap.org/soap/envelope/"。

(2) SOAP 的编码规则的名字空间标志符是"http://schemas.xmlsoap.org/soap/encoding/"。

SOAP 消息中不能包含文档类型声明，也不能包括消息处理指令。

SOAP 使用"ID"类型"id"属性来指定一个元素的惟一的标志符，同时该属性是局部的和无需校验的。SOAP 使用"uri-reference"类型的"href"属性指定对这个值的引用，同时该属性是局部的和无需校验的。这样就遵从了 XML 规范，XML 模式规范和 XML 连接语言规范的风格。

除了 SOAP mustUnderstand 属性和 SOAP actor 属性之外，一般允许属性和它们的值出现在 XML 文档实例或模式中(两者效果相同)。也就是说，在 DTD 或模式中声明一个默认值或固定值和在 XML 文档实例中设置它的值在语义上相同。

6.5.4 SOAP 封装

SOAP 消息是一个 XML 文档，包括一个必须的 SOAP 封装，一个可选的 SOAP 头和一个必需的 SOAP 体。一般谈到 SOAP 消息时就是指这个 XML 文档。一个 SOAP 消息包括以下部分。

在表示这个消息的 XML 文档中，封装是顶层元素。元素名"Envelope"在 SOAP 消息中必须出现，可以包含名字空间声明和附加属性。如果包含附加属性，这些属性必须限定名字空间。"Envelope"可以包含附加子元素，这些也必须限定名字空间且跟在 SOAP 体元素之后。

应用 SOAP 交换信息的各方是分散的且没有预先协定，SOAP 头提供了向 SOAP 消息中添加关于这条 SOAP 消息的某些特征的机制。SOAP 定义了少量的属性用来表明这项特征是否可选以及由谁来处理。元素名是"Header"，在 SOAP 消息中可能出现。如果出现的话，必须是 SOAP 封装元素的第一个直接子元素。SOAP 头可以包含多个条目，每个都是 SOAP 头元素的直接子元素。所有 SOAP 头的直接子元素都必须限定名字空间。

SOAP 体是包含最终接收者想要消息的信息容器。SOAP 为 SOAP 体定义了一个 Fault 元素用来报告错误信息。元素名是"Body"，在 SOAP 消息中必须出现且必须是 SOAP 封装元素的直接子元素。它必须直接跟在 SOAP 头元素(如果有的话)之后，否则它必须是 SOAP 封装元素的第一个直接子元素。SOAP 体可以包括多个条目，每个条目必须是 SOAP 体元素的直接子元素。SOAP 体元素的直接子元素可以限定名字空间。

SOAP 为相互通信的团体之间提供了一种很灵活的机制：在无须预先协定的情况下，以分散但标准的方式扩展消息。可以在 SOAP 头中添加条目实现这种扩展，典型的例子有认证、事务管理和支付。

头元素编码为 SOAP 封装元素的第一个直接子元素。头元素的所有直接子元素称做条

目。

SOAP 体元素提供了一个简单的机制，使消息的最终接收者能交换必要的信息。使用体元素的典型情况包括配置 RPC 请求和错误报告。

体元素编码为 SOAP 封装元素的直接子元素。如果已经有一个头元素，那么体元素必须紧跟在头元素之后，否则它必须是 SOAP 封装元素的第一个直接子元素。

体元素的所有直接子元素称做体条目，每个体条目在 SOAP 体元素中编码为一个独立的元素。

虽然头和体定义为独立的元素，它们实际上是有关系的。体条目和头条目的关系如下：体条目在语义上等同于 actor 属性为默认值且 mustUnderstand 属性值为“1”的头条目。不使用 actor 属性则表示默认的 actor。

SOAP 错误元素用于在 SOAP 消息中携带错误和(或)状态信息。如果有 SOAP 错误元素，它必须以体条目的方式出现，并且在一个体元素中最多出现一次。

6.5.5 SOAP 编码

SOAP 编码格式基于一个简单的类型系统，概括了程序语言，数据库和半结构化数据等类型系统的共同特性。一个类型或者是一个简单的(标量的)类型，或者是由几个部分组合而成的复合类型，其中每个部分都有自己的类型。以下将详细描述这些类型。这一节定义了类型化对象的序列化规则，它分两个层次。首先，给定一个与类型系统的符号系统一致的模式，就构造了 XML 语法的模式。然后，给定一个类型系统的模式和与这个模式一致的特定的值，就构造了一个 XML 文档实例。反之，给定一个依照这些规则产生的 XML 文档实例和初始的模式，就可以构造初始值的一个副本。

XML 允许非常灵活的数据编码方式。SOAP 定义了一个较小的规则集合。编码规则请参见 SOAP 协议。

6.5.5.1 简单类型

SOAP 采用了“XML Schema Part 2: Datatypes”规范“Built-in datatypes”节中的所有类型作为简单类型，包括值和取值范围。例如表 6.3。

表 6.3 SOAP 简单类型

类　型	例　子
Int	58502
Float	3141592658979E+1
negativeInteger	-32768
string	Louis “Satchmo” Armstrong

在 XML Schema 规范中声明的数据类型可以直接用在元素 schema 中，也可以使用从这些类型衍生的新类型。一个模式和对应的具有这些类型的元素的数据实例如下所示：

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
```

```
    <simpleType base="xsd:string">
      <enumeration value="Green"/>
      <enumeration value="Blue"/>
    </simpleType>
</element>

<age>45</age>
<height>5.9</height>
<displacement> -450</displacement>
<color>Blue</color>
```

所有简单值必须编码为元素的内容，它的类型或者在“XML Schema Part 2: Datatypes”规范中定义过，或者是基于一个用 XML 模式规范提供的机制能衍生出的类型。

如果一个简单值编码为独立元素或异质数组成员，那么有一个对应于数据类型的元素声明将会很方便。因为“XML Schema Part 2: Datatypes”规范包括了类型定义，但是不包括对应的元素声明，SOAP-ENC 模式和名字空间为每个简单数据类型声明了一个元素，如

```
<SOAP-ENC:int id="int1">45</SOAP-ENC:int>
```

6.5.5.2 Compound types 复合类型

SOAP 定义了下列常在程序语言中出现的结构性模式对应的类型：

1. 结构

一个“struct”是一个复合值，它的成员值的惟一区别是 accessor 名称，任意两个 accessor 名称都不相同。

2. 数组

一个“array”是一个复合值，它的成员值的惟一区别是序数位置。

SOAP 也允许结构和数组之外的其他数据的序列化，例如 Directed-Labeled-Graph Data Model 之类的数据中，单个节点有许多不同的 accessor，有些不止出现一次。SOAP 序列化规则不要求底层的数据模型在 accessor 之间区分次序，但如果有这样的次序的话，这些 accessor 必须按照这个顺序编码。

6.5.6 在 HTTP 中使用 SOAP

把 SOAP 绑定到 HTTP，无论使用或不用 HTTP 扩展框架，都有很大的好处：在利用 SOAP 的形式化和灵活性的同时，使用 HTTP 种种丰富的特性。在 HTTP 中携带 SOAP 消息，并不意味着 SOAP 改写了 HTTP 已有的语义，而是将构建在 HTTP 之上 SOAP 语义自然地对应到 HTTP 语义。

SOAP 自然地遵循 HTTP 的请求/应答消息模型使得 SOAP 的请求和应答参数可以包含在 HTTP 请求和应答中。注意，SOAP 的中间节点与 HTTP 的中间节点并不等同，即不要期望一个根据 HTTP 连接头中的域寻址到的 HTTP 中间节点能够检查或处理 HTTP 请求中的 SOAP 消息。

在 HTTP 消息中包含 SOAP 实体时，按照 RFC2376 参照，HTTP 应用程序必须使用内容类型 “text/xml”。

6.5.6.1 SOAP HTTP 请求

虽然 SOAP 可能与各种 HTTP 请求方式相结合,但是绑定仅定义了在 HTTP POST 请求中包含 SOAP 消息。

一个 HTTP 请求头中的 SOAPAction 域用来指出这是一个 SOAP HTTP 请求,它的值是所要的 URI。在格式、URI 的特性和可解析性上没有任何限制。当 HTTP 客户发出 SOAP HTTP 请求时必须使用在 HTTP 头中的这个域。

```
soapaction     = "SOAPAction" ":" [ <"> URI-reference <"> ]
URI-reference = <as defined in RFC 2396 [4]>
```

HTTP 头中 SOAPAction 域使服务器(如防火墙)能正确的过滤 HTTP 中 SOAP 请求消息。如果这个域的值是空字符串(" "),表示 SOAP 消息的目标就是 HTTP 请求的 URI。这个域没有值表示没有 SOAP 消息的目标的信息。

例子:

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
SOAPAction: "myapp.sdl"
SOAPAction: ""
SOAPAction:
```

6.5.6.2 SOAP HTTP 应答

SOAP HTTP 遵循 HTTP 中表示通信状态信息的 HTTP 状态码的语义。例如,2xx 状态码表示这个包含了 SOAP 组件的客户请求已经被成功的收到,理解和接受。

在处理请求时如果发生错误,SOAP HTTP 服务器必须发出应答 HTTP 500 "Internal Server Error",并在这个应答中包含一个 SOAP Fault 元素表示这个 SOAP 处理错误。

6.5.6.3 HTTP 扩展框架

一个 SOAP 消息可以与 HTTP 扩展框架一起使用以区分是否有 SOAP HTTP 请求和请求的目标。

使用扩展框架还是普通的 HTTP 涉及到通信各方的策略和能力。通过使用一个必须的扩展声明和"M-"HTTP 方法名前缀,客户可以强制使用 HTTP 扩展框架。服务器可以使用 HTTP 状态码 510 "Not Extended"强制使用 HTTP 扩展框架。也就是说,使用一个额外的交互,任何一方都可以发现另一方的策略并依照执行。

用来表示 SOAP 使用了扩展框架的扩展标志符是:

http://schemas.xmlsoap.org/soap/envelope/

6.5.6.4 SOAP HTTP 举例

例 3 使用 POST 的 SOAP HTTP,

```
POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<SOAP-ENV:Envelope...

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope...
```

例 4　使用扩展框架的 SOAP HTTP

```
M-POST /StockQuote HTTP/1.1
Man: "http://schemas.xmlsoap.org/soap/envelope/"; ns=NNNN
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
NNNN-SOAPAction: "http://electrocommerce.org/abc#MyMessage"
<SOAP-ENV:Envelope...

HTTP/1.1 200 OK
Ext:
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope...
```

6.5.7　在 RPC 中使用 SOAP

设计 SOAP 的目的之一就是利用 XML 的扩展性和灵活性来封装和交换 RPC 调用。这一节定义了远程过程调用和应答的统一表示形式。

虽然可以预计到这种表示形式最可能被用于与 6.5.5 中定义的编码方式相结合，但也可能有其他的表示形式。SOAP 的 encodingStyle 属性可以用来表明方法调用和应答都使用这一节所指定的表示方式。

在 RPC 中使用 SOAP 和 SOAP 协议绑定是紧密相关的。在使用 HTTP 作为绑定协议时，一个 RPC 调用自然地映射到一个 HTTP 请求，RPC 应答同样映射到 HTTP 应答。但是，在 RPC 中使用 SOAP 并不限于绑定 HTTP 协议。

要进行方法调用，以下的信息是必须的：

(1) 目标对象的 URI。

(2) 方法名。

(3) 方法 signature(可选)。

(4) 方法的参数。

(5) 头数据(可选)。

SOAP 依靠协议绑定提供传送 URI 的机制。对 HTTP 来说，请求的 URI 指出了调用的来源 。除了必须是一个合法的 URI 之外，SOAP 对一个地址的格式没有任何限制。

6.5.7.1 RPC 和 SOAP 体

RPC 方法调用和应答都包含在 SOAP Body 元素中,它们使用如下的表示形式:

(1) 一个方法调用用一个结构表示。

(2) 一个方法调用被看做一个单个的结构,每个[in]和[in/out]参数有一个 accessor。结构的名和类型与方法相同。

(3) 每个[in]和[in/out]参数都被看做一个 accessor,这个 accessor 的名和类型与参数的名和类型相对应。它们的出现顺序和方法中定义的参数顺序相同。

(4) 一个方法应答用一个结构表示。

(5) 一个方法应答被看做一个单个的结构,返回值和每个[in]和[in/out]参数有一个 accessor。第一个 accessor 是返回值,之后是参数 accessor,参数 accessor 的出现顺序和方法中定义的参数顺序相同。

(6) 每个参数 accessor 的名称和类型与参数的名称和类型相对应。返回值 accessor 的名称并不重要。同样,结构的名称也不重要。不过,通常在方法名称的后面加上字符串"Response"作为结构的名称。

(7) 方法错误使用 SOAP Fault 元素表示。如果绑定的协议有额外的规则表示错误,则这些规则也必须要遵从。

应用程序可以处理缺少参数的请求,但是可能返回一个错误。

因为返回结果表示调用成功,提示错误表示调用失败,所以在方法应答中同时包含返回结果和错误则是错误的。

6.5.7.2 RPC 和 SOAP 头

在 RPC 编码中,可能会有与方法请求有关但不是正规方法特征的附加信息。如果这样,它必须作为 SOAP 头元素的子元素。

一个使用这种头元素的例子是在消息中传递事务 ID。由于事务 ID 不是方法特征的一部分,通常由底层的组件而不是应用程序代码控制,所以没有一种直接的方法在调用中传递这个必要的信息。通过在头中添加一个给定名字的条目,接收方的事务管理器就可以解析这个事务 ID,而且不影响远程过程调用的代码。

第 7 章　XML 与数据库

本章讨论 XML 与数据库的关系，结合的类型，并在 XML 文件存储与检索中重点介绍了数据库视图与 DTD 的映射方法及其互逆过程。同时介绍了 XML 相关的数据库产品，着重介绍了 SQL Server 2000 和 Oracle8i 对 XML 的支持技术。

7.1　XML 与数据库的关系

在讨论 XML 与数据库关系的问题之前首先需要讨论 XML 本身是不是数据库这一问题。严格地讲，如果"XML"是指 XML 文档，那么答案是否定的。尽管 XML 文档包含了数据，但是如果没有其他软件来处理这些数据，它对于数据库的意义和其他文本文件没有根本区别。但在更为广泛的意义上，当 XML 是指 XML 文档以及所有相关的 XML 的工具和技术时，答案则是肯定的。尽管 XML 本身并不能和数据库挂上钩，但是加上一些其他辅助工具，我们可以把整个 XML 看成是一个数据库系统，XML 文档本身可以看成是数据库中的数据区，DTD 或者模式可以看成是数据库模式设计，XQL 可以看成是数据库查询语言，SAX 或 DOM 可以看成是数据库处理工具。但 XML 还缺少很多在真实的数据库中所必备的内容：有效的存储、索引、安全、事务、数据完备性、多用户访问、触发和多文档查询等。因此在数据量一般、用户较少、性能要求不高的环境下可以把 XML 当做数据库来使用；而如果有许多的用户使用、需要严格的数据完整性并且对性能有很高的要求，XML 就不很适用。

为什么要使用数据库？是需要将原有的数据导出还是要在一个电子商务应用中使用数据库，并将 XML 当做数据传输格式？这些问题的答案都将直接影响到对数据库和中间件的选择。若是在电子商务应用程序中使用 XML 来进行数据传输，这是很好的方案。因为数据具有高度规范的结构，而 XML 中的那些实体和编码对此并不重要。我们关心的是数据而不是这些数据如何在文档中进行物理的存储。如果应用程序相对简单，关系数据库和数据传输中间件可满足需求；如果应用程序庞大而且复杂，则需要一个完全支持 XML 的开发环境。另一方面，如果一个 Web 站点的内容是由一系列 XML 文档构成的，我们不仅要管理这个站点，同时要提供给用户一个搜索该站点内容的机制。而这些都需要借助数据库来实现。数据库和 XML 对于存储数据提供了互补的功能。数据库存储是为了有效地恢复，而 XML 则提供了容易的信息交换方法使得应用之间可以进行互操作。

7.2　XML 与数据库的结合类型

实际上，XML 文档可以分为两大类：以数据为中心或者以文档为中心。在选择数据库时，最重要的判断因素是利用数据库来保存数据还是保存文档。如果保存数据，可以选择面向数据存储的数据库（关系型数据库或者面向对象型数据库）并在数据库和 XML 文档之间相互转换。如果存储文档，那么就需要一个专门设计用来存储文件的内容管理系统。

以数据为中心的文件特点是结构相当规范、数据颗粒度好（数据中最小的独立单元是 PC-

DATA 元素或者是属性)、很少或者没有混合内容。其中同层次元素和 PCDATA 的出现顺序并不重要。典型的例子是 XML 文档包含了销售订单、飞行安排和餐馆菜单等。数据为中心的文档常被用于机器处理,这时 XML 仅是数据传输的手段而已。例如:

```
<Orders>
   <SalesOrder SONumber="123">
      <Customer CustNumber = "543">
         <CustName>ABC Industries</CustName>
         <Street>123 Main St. </Street>
         <City>Chicago</City>
         <State>IL</State>
         <PostCode>60609</PostCode>
      </Customer>
      <OrderDate>02.01.99</OrderDate>
      <Line LineNumber="1">
         <Part PartNumber="123">
            <Description>
               <P><B>Turkey wrench</B><BR />
               Stainless steel, one-piece construction,
               lifetime guarantee. </P>
            </Description>
            <Price>9.95</Price>
         </Part>
         <Quantity>10</Quantity>
      </Line>
   </SalesOrder>
</Orders>
```

以文档为中心的文档的特点是:结构不规范、数据颗粒度更大(最小的独立数据单元是包含有混合内容的元素或者就是整个 XML 文档)以及含有大量的混合内容。其中相同层次的元素和 PCDATA 出现顺序是非常重要的。典型的例子是书、电子邮件、广告以及大多数 XHTML 文档。以文档为中心的文档适用于人类的使用。例如:

```
<Product>
   <Name>Turkey Wrech</Name>
   <Developer>Full Fabrication Labx, Inc. </Developer>
   <Summary>Like a monkey wrech, but nt as big. </Summary>
   <Description>
   <Para>The turkey wrench, which comes in both right- and left-handed versions(skyhook optional), is made of the finest stainless steel. The Readi-grip rubberized handle quickly adapts to your hands, even in the greasiest situations. Adjustment is possible through a variety of custom dials. </Para>
   <Para>You can:</Para>
   <List>
   <Item><Link URL="Order. html">Order your own turkey wrech</Link></Item>
   <Item><Link URL="Wreches. html">Read more about wreches</Link></Item>
```

```
    <Item><Link URL="catalog.zip">Download the catalog</Link></Item>
    </List>
    <Para>The turkey wrench costs just $19.99 and, if you order now, comes with a hand-crafted shrimp hammer as a bonus gift.</Para>
    </Description>
</Product>
```

在现实情况中,以数据为中心的文档和以文档为中心的文档之间的区别并不是很严格。一个以数据为中心的文件(如一张发票),也有可能包含粗颗粒度、不规则的数据(如发票的描述部分)。一个以文档为中心的文件(如用户手册)也可能包含有良好颗粒度、规则的结构化数据(通常是元数据),例如:作者和修订日期。除此之外,让你的文档具有以数据为中心或者以文档为中心的特点有助于判断是关心数据还是文档,这也将决定需要采用什么样的系统。

目前,对 XML 数据库的存储有两种不同的观点:一部分人认为 XML 只有按 XML 本身结构存储的数据库才是 XML 数据库(Native XML Database),另一部分人主张如果能实现对 XML 的存取则它就是 XML 数据库(XML-Enabled Database),也有一部分人主张两者的结合(Hybrid XML Database),下面将分别介绍。

7.2.1 Native XML Database(NXD)

NXD 的最终设计目的是为存储和处理 XML 文档。它的基本存储单元是 XML 文档,通过 XML 相关的标准进行数据库的存储。这种数据库维持原有 XML 文档的数据结构和相关的元数据,而不关心数据的底层存储格式(关系数据库或是面向对象数据库),只能通过 XML 特有的相关技术对数据进行存储。Tamino, dbXML, IXIASOFT XML server 和 X-Hive 采用的是这种方法。IXIASOFT XML server 对原始 XML 文档进行解析,并在原文档的基础上建立相应的索引机制,从而提高了数据库的检索效率。

7.2.2 XML-Enabled Database (XEDB)

XEDB 的基本数据存储单位是 XML 数据(XML 文档所提供的数据),主要是通过增加一个映射层来管理 XML 数据的存储,它是 XML 与数据库之间转换的桥梁。数据首先要与一个明确的格式相匹配,符合要求的才能根据预先定义好的规则映射到数据库(关系数据库或是面向对象数据库)中,但可能会损失一部分元数据和最初的文档结构。同时可以从现有的数据库中动态生成 XML 页面,但不能保证与当初存入的原始页面完全符合。Oracle8i, Microsoft 和许多 XML 工具软件都可以不同程度地支持这种功能。

7.2.3 Hybrid XML Database(HXD)

HXD 根据具体的要求和应用,既可以做 Native XML Database 又可以做 XML Enabled Database, Excelon 和 Ozone 就采用了这种思想。

综上所述,以 XML 文档为中心的方法,保留了文档的原始结构和 XML 原有的优点,存储简单,同时可以将 XML 文档看成对象树,有利于对单个 XML 文档的进一步的数据挖掘。但信息的格式、内容相对繁杂,索引建立庞大复杂。

以数据为中心的方法将 XML 文档的数据进行重新组织,使数据的存储相对规范,有利于在此基础上对信息的进一步应用,比如智能检索、电子商务等,但破坏了原文档的结构,也很难

保证文档恢复到原来的结构，在存入关系数据库中的预先处理工作量比较大，尤其是当XML文档不很规范化时需要对它进行复杂的分解、映射等处理。XML最具吸引力的特性之一的分层结构，被关系数据库映射成关系表，从而将XML结构变成了平面的行和列。遇到大型或复杂文档时，在XML与数据库之间进行来回转换要耗费相当多的处理时间，从而降低了Web页面的生成速度。但电子商务等方面的XML文档可以通过符合指定的DTD进行规范化，同时可以利用数据库技术中成熟的统计、并发事务处理等技术，并且从企业的观点来看选择关系数据库和面向数据库是比较安全的，不愿冒险使用新的Native XML。

现在已经有一些能够很好地处理XML的XML-Enabled数据库，并且是由关系数据库和面向对象数据库担此重任。这些数据库在收到XML后，将其分解为字段并按通常的方式存储它们，当检索XML时，这些字段再被重新组织成XML文档。麻省Reading的Xyvision企业解决方案公司研制的Content@XML是一套内容管理系统，它可以在任何一种流行的关系数据库中存储XML文件。其好处是可以开展基于内容的协同工作，进行多通道内容输出。一家技术出版商选用了Content@XML，认为XML将他们两周的工作压缩到几分钟，"该系统接收的是XML材料，给出的却是你想要的任何格式的结果"。Lotus公司的Domino数据库也可以处理XML，而且其XML Toolkit甚至可以使用户像在Native XML数据库中一样创建和处理内容。在关系数据库中处理XML数据时，可用第三方中间件进行转换，其中比较好的是XML-DBMS，这是一种基于Java Database Connectivity(JDBC)的工具，可以在XML文档和数据库之间传输数据。

另一方面，第一个商用Native XML数据库是位于美国加州的Software AG公司开发的Tamino。除了可以存储和访问XML外，Tamino还具备多项功能，包括Open Database Connectivity、符合Unicode要求、HTTP通信及处理非XML数据的能力。Gartner Group的一份报告指出："Tamino特别适用于需要从多种不同平台和格式整合信息并向业务伙伴或客户散发这些信息的机构。"Tamino拥有直接XML检索和特殊检索的能力，其查询语言强大而简短，可进入任意深度，使SQL相形见绌。Native XML数据库的主要缺陷之一是性能问题。有人认为，当所搜寻的信息位于大文档的末尾时，由于缺乏其他机制，Native XML数据库只能艰苦跋涉到最后，而关系数据库和面向对象数据库则可以将文档分成小块同时进行搜索，速度当然要快得多。当然，上述困难并非无法克服，只要在存储时给各文档分别建立索引机制。Tamino就具有这种索引能力，从而弥补了大型文档搜索的不足，另外，Native XML存储消除了不必要的转换操作。目前，Tamino有Windows NT、Windows 2000、Solaris和SCO UNIX等版本，将来还要出Linux和一些大型机版本。目前，许多主流的数据库厂商都在把XML支持结合到其产品中，或者提供可在其数据库中使用XML的工具。IBM提供了XML Extender for DB2，以允许用户在DB2中存储XML文档，并提供一些新功能协助用户处理XML文档；Microsoft的SQL Server 7.0也进行了一定的XML的扩充，SQL将来要加入XML输出选项，用以向其他系统传送信息。Oracle也拥有功能强大的XML索引引擎。总之，XML的需求正在扩大，新的应用包括采用XML标签的Internet搜索引擎、必须快速输出结果的电子商务系统、带XML标签的电子数据交换、数据重复使用和内容个性化。作为处理上述应用的一环，XML数据库的需求也将快速增长。

7.3 存储和检索数据

在以数据为中心的文档中的数据内容可能来自数据库(将数据导出为XML格式)，也可

能是 XML 文档(将数据存储在数据库中)。前者的例子是在关系型数据库中存储的大量现有数据(或称遗产数据);后者的例子是将数据作为 XML 发布在 Web 中,而且想要在数据库中进行存储以进行更多的处理。因此,根据需求可能需要将 XML 文档转移到数据库的软件,也可能需要从数据库转移到 XML 文档的软件,或者两者都支持。

7.3.1 转移数据

将数据存储在数据库中时,经常需要丢弃大量与文档有关的信息,例如文档名称和 DTD,同时还有其物理结构,例如实体的定义和使用,属性值和同层元素的顺序,二进制数据的存储方式(Base64 编码、未解析实体或其他方式),字符数据段和其他的编码信息。当从数据库中检索数据时,生成的 XML 文档结果除了非预定义实体 lt("<"),gt(">"),amp("&"), apos("'"), quot(""")不包含任何 CDATA 或实体引用。而同层元素和属性的出现顺序也常常就是从数据库中返回的数据的次序。

例如,假设需要用 XML 作为数据格式把一张销售单从一个数据库中转移到另一个数据库中。在这种情况下,在 XML 文档中并不关心销售单的编号是保存在销售单的日期的前面还是后面,也不用关心是否将顾客的名称保存在字符数据(CDATA)段还是作为一个外部实体,或者直接当成一个 PCDATA。最重要的在于相关的数据是从第一个数据库转移到第二个数据库中。这样,这个数据传输软件就需要考虑数据的层次结构(该结构将销售单的有关内容进行了分组),而其他则不必过多考虑。

忽略文档信息以及其物理结构的后果之一是文档"逆反回归"的不一致效应,即将一个文档的数据存储在数据库中,然后根据这些数据重新组织成新的文档。即便是根据标准格式处理,得到的也常常是和前面不同的文档。这是否可以接受要取决于需求,而且也将影响到对数据库和数据传输中间件的选择。

7.3.2 从文档结构到数据库结构的映射类型

为了在 XML 和数据库之间传输数据,需要在文档结构和数据库结构之间进行相互的映射。这样的映射通常分为两大类:模板驱动和模型驱动。

7.3.2.1 模板驱动的映射

在以模板驱动的映射中,没有预先定义文档结构和数据库结构之间的映射关系,而是使用将命令语句内嵌入模板的方法,让数据传输中间件来处理该模板。考虑下面的模板,在<SelectStmt>元素中内嵌了 SELECT 语句:

```
<? xml version="1.0"? >
<FlightInfo>
        <Intro>The following flights have available seats:</Intro>
        <SelectStmt>SELECT Airline, FltNumber, Depart, Arrive FROM
Flights</SelectStmt>
        <Conclude>We hope one of these meets your needs</Conclude>
</FlightInfo>
```

当数据传输中间件处理到该文档时,每个 SELECT 语句都将被各自的执行结果所替换,得到下面的 XML 格式:

```
<? xml version="1.0"? >
<FlightInfo>
    <Intro>The following flights have available seats:</Intro>
    <Flights>
        <Row>
            <Airline>ACME</Airline>
            <FltNumber>123</FltNumber>
            <Depart>Dec 12, 1998 13:43</Depart>
            <Arrive>Dec 13, 1998 01:21</Arrive>
        </Row>
        ...
    </Flights>
    <Conclude>We hope one of these meets your needs</Conclude>
</FlightInfo>
```

这种以模板驱动的映射可以相当地灵活。有些产品可以允许在任何结果集合中替换想要的内容(包括在 SELECT 中使用参数),而不是像上面的例子中简单地格式化结果。另外它还支持使用编程来进行构造,例如循环和条件判断结构。还有一些支持 SELECT 语句的参数化,例如通过 HTTP 来传递参数。

7.3.2.2 模型驱动的映射

在以模型驱动的映射中,利用 XML 文档结构对应的数据模型显式或隐式地映射成数据库的结构,而且反之亦然。它的缺点是灵活性不够,但是却简单易用。这是因为它是基于具体的数据模型来进行映射的,通常能够为用户实现很多转换工作。由于将数据从数据库转换成 XML 的结果依照了单个模型,因此在这种方式下通常结合 XSL 来提供模板驱动的系统中所具有的灵活性。

在 XML 文档中的数据视图通常有两种模型:表格模型和特定数据对象模型。有时候也可能会出现其他的模型。通过采用 ID 和 IDREF 属性,一个 XML 文档可以用于一个指定的模型。不过,很多现有的中间件并不支持这些模型。

1. 表格模型

许多中间件软件包都采用表格模型在 XML 和关系型数据库之间进行转换。它把 XML 的模型看成是一个单独的表格或者是一系列的表格。也就是说,XML 的文档结构和下面的例子相类似,其中在单个表格的情况下,<database>并不出现:

```
<database>
    <table>
        <row>
            <column1>...</column1>
            <column2>...</column2>
            ...
        </row>
        ...
    </table>
```

```
...
</database>
```

其中的术语"table"可理解为单个的结果集(当从数据库向 XML 中转换数据时),或者是一个单独的表格或可更新的视图(当从 XML 向数据库转换数据时)。如果数据需要来自多个结果集(当数据来自数据库中时)或者 XML 的文档包含有更深层次的嵌套元素,那么类似的转换几乎是不可能的。

2. 特定数据对象模型

XML 文档中第二种普遍的数据模型是特定数据对象的树型结构。在该模型中,元素类型通常对应对象,而 XML 中的内容模型、属性和 PCDATA 则对应对象的属性。这种模型直接映射成面向对象的数据库和层次型数据库,当然借助于传统对象-关系映射技术和 SQL 3 对象视图也可以映射成关系数据库。要注意的是,这种模型并不是文档对象模型(DOM)。DOM 是对文档本身进行建模,而不是对文档中的数据。DOM 用来在关系型数据库的基础上建立内容管理系统。

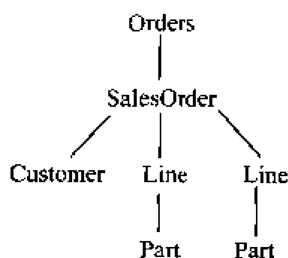


图 7.1 树型结构图

例如,上面的销售订单文档就可以看做是由五个类所组成的树型结构。如图 7.1 所示,包括 Orders, SalesOrder, Customer, Line 和 Part 类。

当把一个 XML 文档建模为一棵特定数据对象树时,就没有必要要求元素一定要对应于对象。如果一个元素只包含 PCDATA ,如销售订单文档中的 CustName 元素,它可以当做一个属性进行处理,因此属性只包含单一的、标量型数值。有时将混合元素或元素内容模型化成属性也是非常有用的。一个现成的例子就是在销售订单文档中对 Description 元素的处理:尽管它在 XHTML 的格式中有混合内容,但是将 Description 元素看做单个的属性会更有用些,因为它的组成部分本身并没有什么意义。

7.3.3 从数据库的结构生成 DTD 及其互逆过程

DTD(Document Type Descriptors)即文档类型定义,它为文档结构定制了一套规则,规定了 XML 文档中的元素类型、属性、文档中的实体及相互关系等。如果将 DTD 文档映射到关系数据库视图,同时生成相应的映射模型,就可以使符合 DTD 的 XML 文档根据映射模型存入关系数据库中。因而如何将一个 DTD 文档映射到关系数据库视图,从而生成具体的映射实体是问题的关键所在。

从 DTD 文档映射到关系数据库视图可分为两个步骤:

(1) 将 DTD 映射到对象视图。

(2) 将对象视图映射到数据库关系视图。

7.3.3.1 基本映射

1. DTD 到对象视图的映射

DTD 中定义的元素(Element)对与该 DTD 相关的 XML 文档中的元素类型进行了说明。DTD 中的元素内容可以分为空、元素、任意和复合 4 种元素类型,空元素中不包含 PCDATA 或子元素,但可以有属性。元素内容模型为仅包含 PCDATA 内容的元素类型,又称为简单元素类型。首先,将简单元素类型映射为标量类型(Scalar Type),该类型的值是单一的和不可分割的数据单元,比如:Java 中的 String、float 类型。元素内容模型中包含子元素、混合内容或属性声明的元素称为复杂元素类型。然后,将复杂元素类型映射为类,其内容模型中的每一元素

类型映射为该类的属性，最后将元素的属性映射为类的属性。

例如图 7.2 所示：简单元素类型 B、D、E 和属性 F 都映射为 String 标量类型，复杂元素类型 A、C 映射为类 A 和 C。由于 C 也映射为一个类，所以将 A 内容模型中的 C 映射为 A 的属性，该属性类型为指向参考类 C 的一个对象。

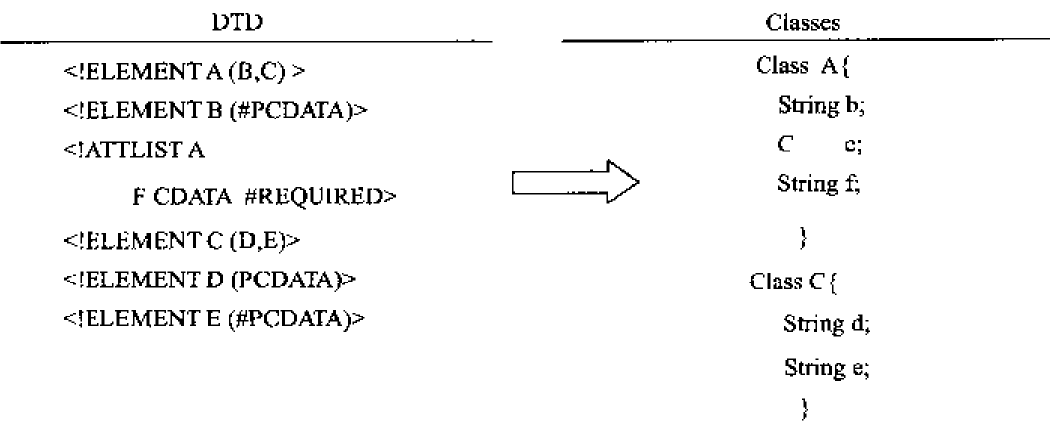


图 7.2 DTD 到对象视图的映射实例

2. 对象视图到数据库关系视图的映射

对象模型到关系模型的映射方法大体可分为三种，如表 7.1 所示。

(1) 对整个类层次仅使用一个表。所有的类和属性都映射到一个表中。这种方法的好处是实现简单。缺点是整体的耦合性太强，对于某一个类增加一个属性就必须对整个表增加一个列，这使得在该属性出现错误会影响整个类层次而不仅仅是影响该子类，并占用了数据库大量的空间。

(2) 对于每一个具体的类(如：Employee，Customer)对应一个表。每一个表包括属性和该类所继承的属性。这种方法的主要优点是每一个类单独存储到一个表中，实现相对简单。缺点是当修改某个类时也必须对应修改它的子类，完整性维护较困难。

(3) 每一个类(如：Person、Employee、Customer)对应一个表，类的属性对应一个列，并具有 OID 属性列。它的最大好处是与面向对象概念一致，易于修改和维护，比如增加超类、子类等。缺点是数据库中会增加很多额外的表，增加的表用于维护每个类对应的表之间的关系。图 7.3 是一个简单的类层次 UML 类图及其三种类层次映射方法的实例。

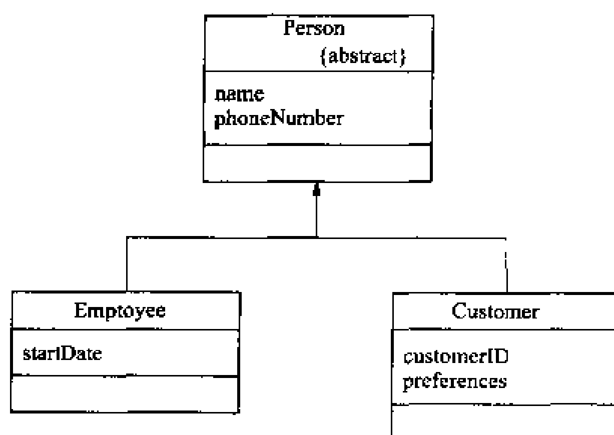
图中的 OID 为对象标识符，它相当于关系数据库中的键值，惟一标志 XML 文档对象树的每一个对象，因而非常重要，它取值惟一但无具体含义。

表 7.1 类层次映射方法比较表

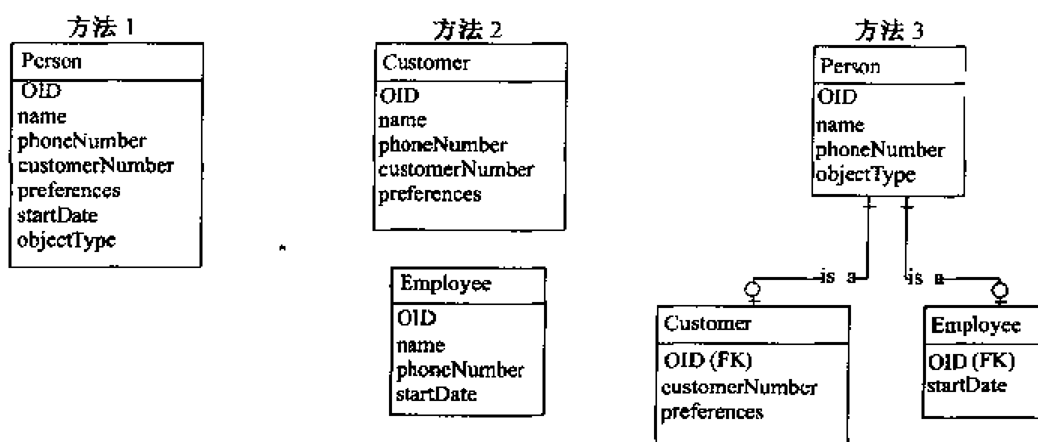
考虑因素	对整个类层次仅使用一个表	每一个具体的类对应一个表	每一个类对应一个表
实现简易性	简单	中等	困难
数据存取的简易性	简单	简单	中等/简单
耦合度	很高	高	低
数据存取速度	快	快	中等/快
支持多态性	中等	低	高

我们采用第三种方法。具体步骤如下：

将类映射到表，具有标量类型的属性映射到表中的列：类名→表名，标量属性名→列名；对



(a)一个简单的类层次的 UML 类图



(b)三种映射方法的实例

图 7.3 映射类层次的三种策略

每个类所对应的表添加主键列，列名为表名_pk；对具有参考关系的两个类所对应的表添加主键/外键关系（若已存在主键列则不用另外生成）：若两个类为一对一关系，则主键列可以在两个类对应的任何一个表之内，同时在另一个表中添加外键列。若两个类为一对多的关系，则主键列须添加在"一"所对应类的表中，另一个表添加外键列。主键列的列名为：表名_pk；外键列的列名为：（主键列所在的）表名_fk。图 7.4 为映射实例。

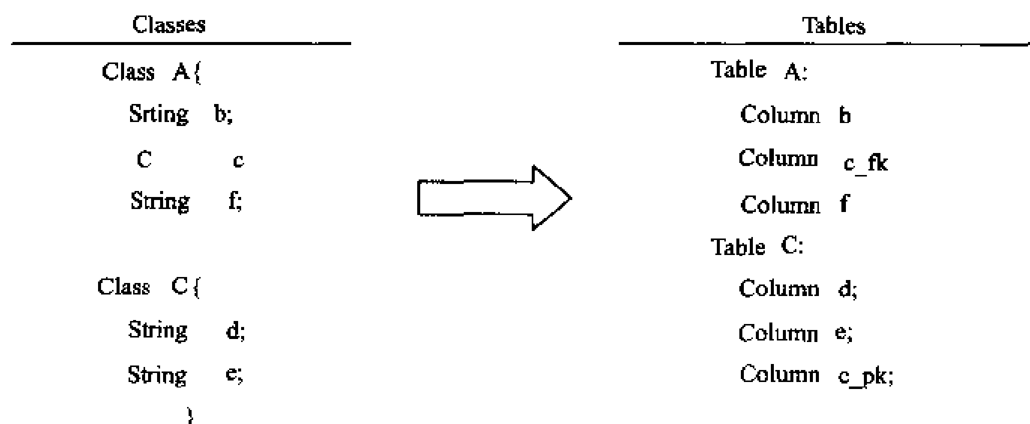
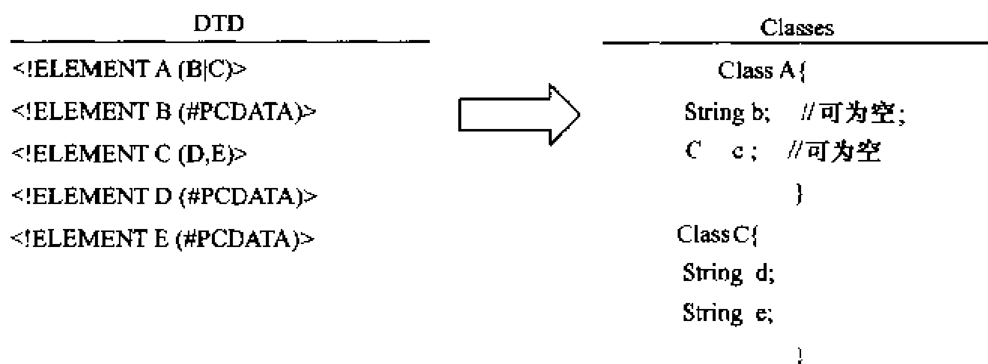


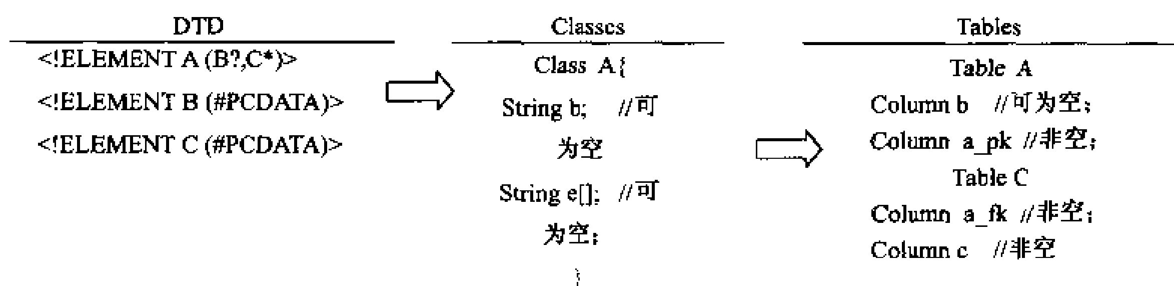
图 7.4 对象视图到关系视图的映射实例

7.3.3.2 复杂内容模型映射

顺序运算符“,”(表示严格顺序)连接的元素类型映射为类的属性,“|”(表示选择)连接的元素类型所映射的类的属性和相应的表的列可以为空。

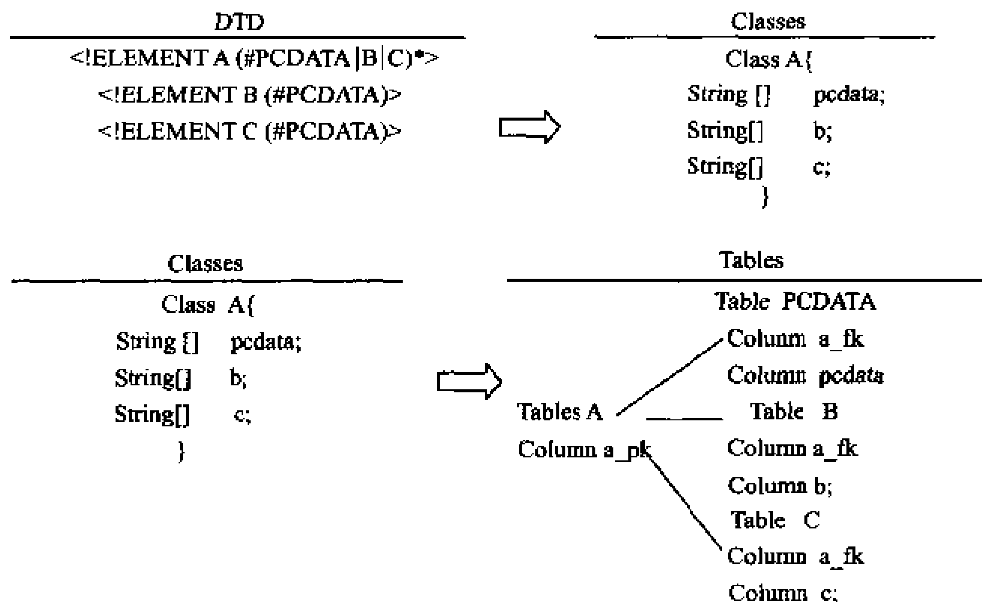


元组运算符“*”(表示零个或多个)、“+”(一个或多个)修饰的元素类型映射为属性,由于其属性值为未知的数组,因而我们将该属性映射为一个属性表。该属性表通过外键与类表中的主键相关联。“?”(表示可选的,零个或一个)修饰的元素类型映射为可为空的属性。



7.3.3.3 混合内容映射

如果一个类中同时有 PCDATA 类型和其他属性,则将 PCDATA 可看做其他子元素,映射为可为空的未知数组,再映射为属性表。该表通过外键与类表中的主键相关联。

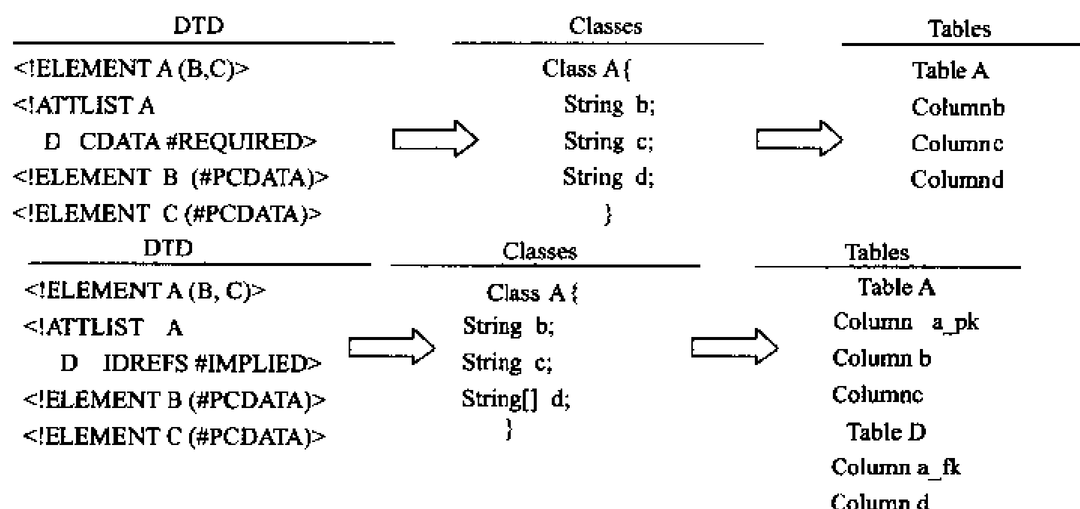


7.3.3.4 属性映射

如前所述,元素类型的属性映射为类的属性。下面进行进一步讨论。

1. 单值和多值属性映射

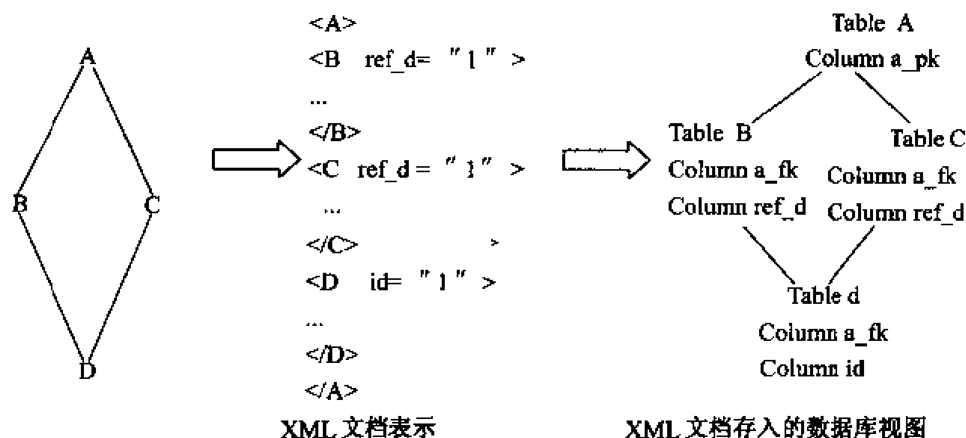
DTD中有两类属性:单值属性(CDATA、ID、IDREF、NMTOKEN、ENTITY、NOTATION,枚举)映射为类的单值属性,再映射为列;多值属性(IDREFS、NMOTOKENS,ENTITIES)映射为类的多值属性,再映射为属性表。例如:



2. ID/IDREF(s)映射

属性类型ID在一个XML文档中惟一标识一个元素;属性类型IDREF、IDREFS是对某些具有ID属性的元素值的引用,这些元素的ID属性值必须与IDREF属性值相同。

例如:



XML 文档表示　　　　XML 文档存入的数据库视图

值得注意的是在数据库中存储ID/IDREF时,ID值只保证在指定的XML文档中的惟一性。因而,当将多个文档中的数据存入数据库中的同一个表中时,不能保证ID值的惟一性。这可以通过将该属性类型映射为两列,一列惟一标识XML文档,另一列记录ID值,或者对ID值本身加上一个惟一值的前缀来解决。目前,大多数中间件都没有将ID/IDREF与其他属性类型区别对待。

总之,对象-关系映射对所有XML文档适用,并允许非XML应用访问数据库中的数据。它不仅对数据为中心的文档是一个很好的选择,而且也是许多中间件、XML-Enabled关系数据库和很多XML-Enabled对象服务器的基本模型。当然,对象-关系映射方法对于普通的文

档并非是很好的选择。具体表现在：当处理混合内容的文档时处理效率不高，与表映射方法一样不能记录文档的物理结构、注释、处理指令等信息。

7.3.3.5 从 DTD 中产生数据库视图算法

数据库视图的产生是通过读人 DTD 文档并处理每个元素类型：

(1) 对复杂元素类型产生带有主键列的类表。

(2) 当不处理内容模型时简单元素类型被忽略。

处理内容模型：

(1) 对仅出现一次的简单元素类型产生列，如果有元组运算符（“?”），则该列可为空。

(2) 对重复出现的简单元素类型产生带有外键的属性表。

(3) 对复杂元素类型在其间接类表中产生外键。

(4) 对混合内容中的 PCDATA 产生带有外键的属性表。

处理属性：

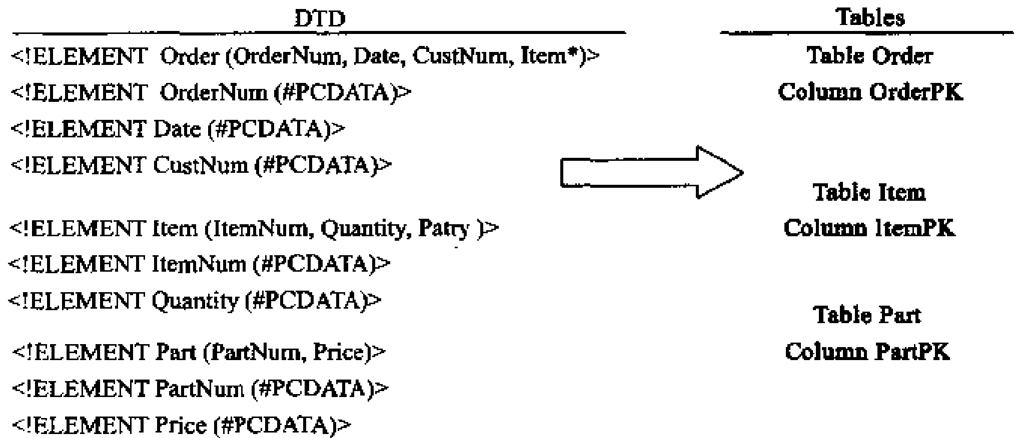
(1) 单值属性产生列；如果属性是可选的，则该列可为空。

(2) 多值属性产生带有外键的属性表。

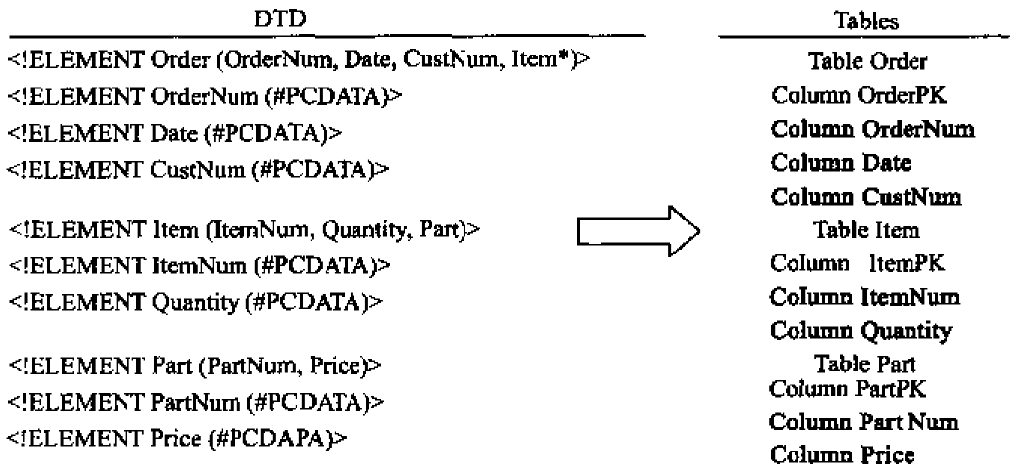
(3) 如果属性有默认值，则将其作为列的默认值。

举例说明处理过程如下。

步骤 1：对复杂元素类型产生带有主键列的类表。



步骤 2：对仅出现一次的简单元素类型产生列。



步骤 3:对复杂元素类型在其间接类表中产生外键。

DTD

```
<!ELEMENT Order (OrderNum, Date, CustNum, Item*)>
<!ELEMENT OrderNum (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT CustNum (#PCDATA)>

<!ELEMENT Item (ItemNum, Quantity, Part)>
<!ELEMENT ItemNum (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>

<!ELEMENT Part (PartNum, Price)>
<!ELEMENT PartNum (#PCDATA)>
<!ELEMENT Price (#PCDAPA)>
```

⇨

Tables

```
    Table Order
Column OrderPK
Column OrderNum
Column Date
Column CustNum
    Table Item
Column   ItemPK
Column ItemNum
Column Quantity
Column OrderFK
    Table Part
Column PartPK
Column Part Num
Column Price
Column PardFK
```

值得注意的是,产生的视图不可能与人工写出的一样,该算法不能够确定 Item 与 Part 是多对一关系;不可能知道 OrderNum 和 PartNum 可以作为主键;不能够决定数据类型和列的长度,但 XML 模式将可以解决后者。

7.3.3.6 从数据库视图中产生 DTD

DTD 的产生是通过单一的"根表"或"根表"的集合开始产生:

(1) 每一个根表(Root Table)都产生一个单一序列的带有元素内容的元素类型。

(2) 根表中的每一个数据列(非键值)都按序列产生一个仅有 PCDATA 内容的元素类型。

主键和外键处理如下:

(1) 间接表(Remote Table)处理方法同根表。

(2) 将对间接表的元素类型引用加入序列。

(3) 如果是主键,引用是可选的和可重复的(* 元组运算符),因为并不能保证在外表中一定存在一行或是仅存在一行。

(4) 如果是主键,主键中的每一列都可选的产生仅有 PCDATA 的元素类型,同时将对这些元素的引用添入序列中,这仅对主键列中包含数据的这种情况起作用。

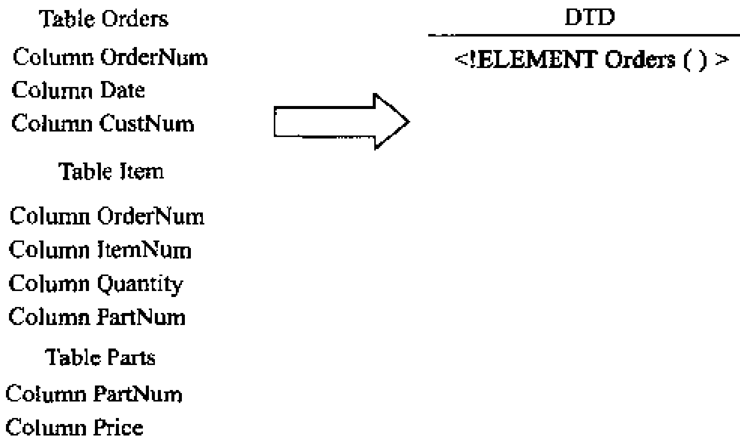
(5) 如果键值是外键并且可为空,则引用是可选的(? 元组运算符)。

举例说明如下:

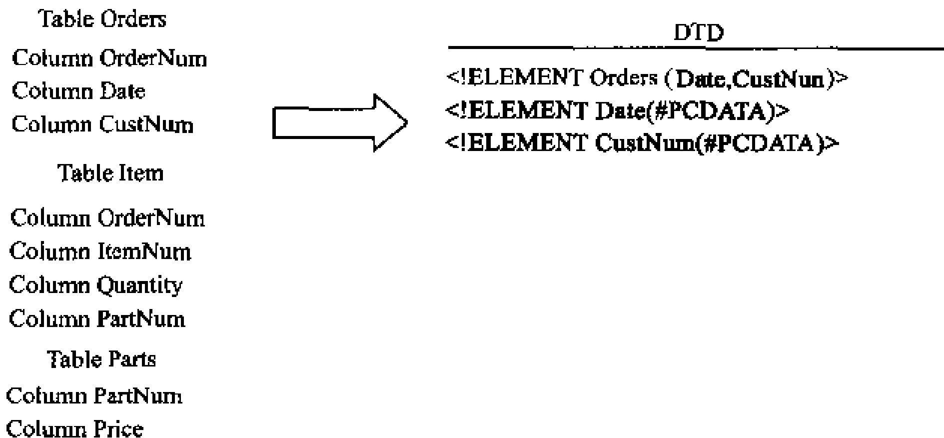
```
    Table Orders
Column OrderNum
Column Date
Column CustNum
    Table Item
Column OrderNum
Column ItemNum
Column Quantity
Column PartNum
    Table Parts
Column PartNum
```

Column Price

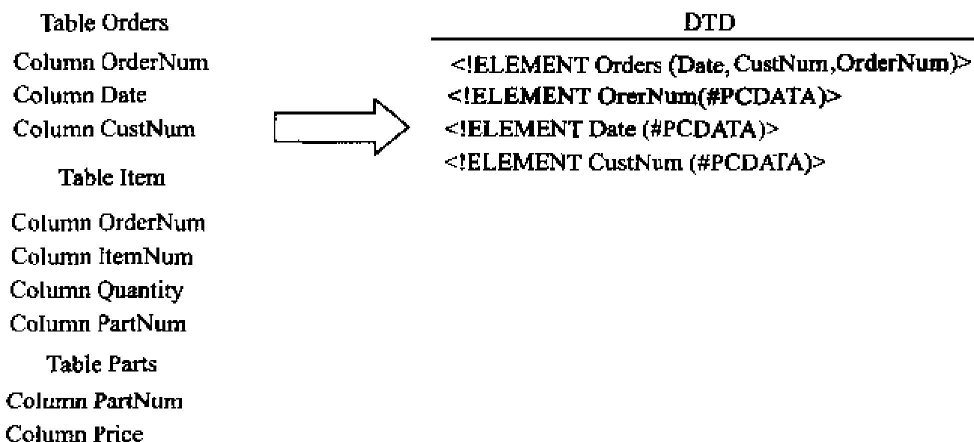
步骤 1:产生根表(Orders)的元素类型。



步骤 2:对数据列(Date 和 CustNum)生成仅有 PCDATA 的元素,并在 Orders 元素的内容模型中加入这些元素的引用。



步骤 3:对主键列(OrderNum)生成仅有 PCDATA 的元素,并在内容模型中加入对它的引用。



步骤 4:根据 Orders 中的主键列找出表 Items,并添加 Items 元素,加入内容模型中。

Table Orders
- Column OrderNum
- Column Date
- Column CustNum

Table Item
- Column OrderNum
- Column ItemNum
- Column Quantity
- Column PartNum

Table Parts
- Column PartNum
- Column Price

DTD

```
<!ELEMENT Orders (Date,CustNum, OrderNum,Items*)>
<!ELEMENT OrderNum (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT CustNum (#PCDATA)>

<!ELEMENT Items ( )>
```

步骤 5：用同样的方法处理间接表中的数据列和主键列。

Table Orders
- Column OrderNum
- Column Date
- Column CustNum

Table Item
- Column OrderNum
- Column ItemNum
- Column Quantity
- Column PartNum

Table Parts
- Column PartNum
- Column Price

DTD

```
<!ELEMENT Orders (Date, CustNum, OrderNum,Items*)>
<!ELEMENT OrderNum (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT CustNum (#PCDATA)>

<!ELEMENT Item(ItemNum,Quantity)>
<!ELEMENT ItemNum(#PCDATA)>
<!ELEMENT Quantity(#PCDATA)>
```

步骤 6:增添外键相关的表 Parts 的元素类型。

Table Orders
- Column OrderNum
- Column Date
- Column CustNum

Table Item
- Column OrderNum
- Column ItemNum
- Column Quantity
- Column PartNum

Table Parts
- Column PartNum
- Column Price

DTD

```
<!ELEMENT Orders (Date,CustNum, OrderNum,Items*)>
<!ELEMENT OrderNum (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT CustNum (#PCDATA)>

<!ELEMENT Item (ItemNum, Quantity,Parts)>
<!ELEMENT ItemNum (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>

<!ELEMENT Parts( )>
```

步骤 7:处理外键表 Parts。

Table Orders

Column OrderNum
Column Date
Column CustNum

Table Item

Column OrderNum
Column ItemNum
Column Quantity
Column PartNum

Table Parts

Column PartNum
Column Price

DTD

```
<!ELEMENT Orders (Date,CustNum, OrderNum,Items*)>
<!ELEMENT OrderNum (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT CustNum (#PCDATA)>

<!ELEMENT Item (ItemNum, Quantity,parts)>
<!ELEMENT ItemNum (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>

<!ELEMENT Parts ( PartNum,Price)>
<!ELEMENT PartNum(#PCDATA)>
<!ELEMENT Price(#PCDATA)>
```

遗憾的是，这些过程还存在着一些缺陷。例如，DTD 中没有方法预先准确地规定数据类型或者字段长度。因为任何的预先定义(例如通过读取一个示例文档)在读取其他“类型”的文档或者其他文档中包含有超过字长内容的文档时就会产生错误。(长久之策是使用 XML 视图文档的数据类型。)当从一关系型结构生成 DTD 时，没有办法预先判断子元素“应该”出现的顺序或者字段(如数据库内部的行标识)是否该进行完全转换。在以上两种情况中都可能产生命名的冲突。

尽管有这样那样的缺陷，但是这些方法仍然能够很好地奠定在关系型结构和 DTD 之间互相转换的起点。

7.4 XML 数据库产品简介

根据 Ronald Bourret 在 XML Database Products 一文中的描述，XML Database 中包含有 7 种类型的产品，分别为：

(1) 中间件(Middleware)。

(2) XML-Enabled Databases，比如 Oralce 和微软都宣称它们最新的数据库产品能够和 XML 进行无缝的衔接。

(3) Native XML Database。

(4) XML 服务器(XML Servers)。

(5) XML 应用服务器，比如 IBM 的 WebSphere。

(6) 内容管理系统(Content Management Systems)。

(7) 可持久化的 DOM 实现(Persistent DOM Implementations)。

下面我们对每一种类型进行具体的说明和介绍。

1. 中间件

所谓中间件就是用来在 XML 文档和数据库之间进行处理和转化的软件。主要应用于以数据为中心的应用，它可以用各种各样的语言编写，一般来说它需要用到 ODBC，JDBC 或 OLE DB。尽管它可以通过 Internet 进行数据的传输，但是一般都是通过 Web 服务器来实现数据的传输。

下面我们需要考虑当把 XML 文档存储到数据库中，如何选择适合应用程序的中间件。

实际上，在我们选择中间件的时候，我们要考虑下面一些因素：

(1) 数据类型：XML 不支持数据类型，也就是说，在 XML 文档中的所有数据都是文本，即使数据本身代表了另外一种数据类型，比如日期或者整数。通常数据传输中间件将把数据转化为其他类型。

(2) 二进制数据处理：有两种通常的方法存储 XML 文档中的二进制数据：未经过解析的实体(Unparsed Entities)和 Base64 编码。

(3) Null 类型处理：在关系数据库世界中，Null 表示该数据不存在，它和 0 或者是空字符串是不一样的。当然，XML 也支持 Null 的概念。如果一个可选择的元素类型或者属性是 Null 的话，它就不包括在这个文档里面。当映射一个 XML 文档的结构到数据库或者根据数据库内容生成 XML 文档的时候，你需要考虑可选元素类型、属性和可以为 Null 的列之间的映射。

(4) 字符集：一个 XML 文档可以包含任何 Unicode 字符，但遗憾的是，许多数据库并不支持 Unicode。因此如果数据包括非 ASCII 字符的时候，需要注意数据库和中间件对这些字符的处理。

(5) 关于 XML 中的处理指令：处理指令并不是 XML 文档中的数据，因此中间件就很难决定如何存储它们。所以在选择中间件的时候，要看它们对处理指令的处理情况。

(6) 标记存储：注意不同的中间件对标记的处理是不一样的。而且在数据库中的存储模式也不同，见下面的例子：

```
<description>
<b>Confusing example:</b>
</description>
```

在数据库中存储的形式如下：

```
<b>Confusing example:</b> <foo/>
```

这主要是因为数据库不能识别<b>和<foo>是标记还是文字。

典型的中间件有：

(1) ADO。它可以实现数据库和 XML 文档之间的双向转换。XML 支持把一个记录集合对象保存为一个 XML 文档，它也能够把一个 XML 文档作为一个记录集合进行处理。这样就在 XML 文档和数据库之间提供了一个桥梁。这种映射实际上是模型驱动的，数据可以被看成是一棵对象树，一个具有嵌套结构的树可以作为一个嵌套的记录结果集被显示，反之也是如此。另外，如果记录集合数据有变化的话，可以反映到相应的 XML 文档中，而 XML 文档中内容的变化也可以导致数据库内容的变化。

(2) ASP2XML。主要是通过一个 COM 对象实现 XML 文档和基于 ODBC 或 OLE DB 的数据库之间数据的传递。该产品实际上是基于模型驱动的，把 XML 文档看成是一个单一的表。当把数据从数据库传递到 XML 文档的时候，用户指定一个 SELECT 语句，输出包含 ASP2XML 专用的标签。当把 XML 文档的数据传递到数据库的时候，XML 文档必须包含 ASP2XML 专用的标签。该 COM 对象是支持自动化的，也就是说它能够在脚本语言中使用，比如 ASP。

2. XML-Enabled Databases

数据库提供了扩展的功能，能够在 XML 文档和数据库之间进行数据的传输。通常是设计成为能够存储和提取以数据为中心的文档。一般来说是把 XML 文档进行解析以后，存储到相应的表格中。当然，也可以存储以文档为中心的文档，也就是说把整个文档作为一个单一

的表中的一个字段，然后通过文本检索机制进行查询。因为许多数据库现在能够把内容发布到网站上，基于 XML 的数据库和 XML 服务器之间的差别就变得很模糊。

典型的产品有：微软的 SQL Server 2000。

3. Native XML Database

典型的产品有斯坦福大学开发的 Lore 系统，它的数据库实际上是一个半结构化的数据库。半结构化数据库，其中既有一些结构化的信息，也有非结构化的信息。XML 本身就是一个非常好的半结构化数据模式的例子，它本身是自描述的，包含了很多元数据。而且它同时可以扩展或增加新的元数据（或者说是新的字段）。

Lore 是用来存储半结构化数据的数据库。它最开始是用来存储 HTML 文档数据的，但现在已经可以被用来作为 XML 数据库。它包括了一个查询语言（Lorel），多索引机制，查询优化器，多用户支持，日志记录和恢复等功能，并且能够导入外部数据。由于 Lore 支持半结构化处理，所以它也能够存储没有 DTD 定义的 XML 文档。

4. XML 服务器

一般认为，XML 服务器是一个提供数据服务的平台，而这里的数据就是以 XML 形式出现的。这些数据主要是为分布式应用程序服务的，比如电子商务和 B2B 应用等。XML 服务器通常包括了一个完整的应用开发环境，并通过各种数据存储方式来使应用程序可以方便地获取和使用这些数据。存储的数据包括传统的数据库数据、电子邮件信息和文件系统等。我们知道传统的 Web 服务器都是基于 HTML 文本进行信息传送的，随着 XML 技术的出现，对于基于 XML 的 Web 服务器的需求也就产生了。那么到底什么是 XML 服务器呢？准确定义 XML Server 这个概念是困难的，因为它是一个崭新的，而且概念广泛的事物。虽然已经有许多产品称自己为 XML Server，比如 DataChannel 公司的 DataChannel Server 4.1，Software AG 公司的 Tamino，Excelon 公司的 excelon，但在应用的范围上以及功能上，每种产品都各不相同。因此这里也不为 XML Server 进行定义，而是归纳这几种产品共同拥有的一些特点，以说明的方式来向大家解释 XML Server 这个概念。简单地讲，XML Server 是一个提供数据的平台，它能够以 XML 文档的形式与分布式的应用进行数据交互。比如电子商务这一类的应用。这听上去和传统的数据库非常的相似，它和数据库一样提供数据的存储与提取功能，但数据的格式是基于 XML 的，因此在数据的处理方面，所用的是和传统数据库完全不同的技术了。

所以 XML Server 被认为是 XML Database 其中的一种。XML-enabled Server 则相对来讲比较容易理解，因为它本质上就是一个 Web Server。对于客户端而言，浏览 Web，不会感觉它和传统 Web Server 有太大的区别，但 Server 端在对请求（Request）的处理方法上，XML-enabled Web Server 和传统 Web Server 是两种完全不同的方式。这是由 XML 文档与 HTML 文档本身的特性所决定的。XML 文档是一种以数据为中心的文档，XML 文档本身没有表示格式化的信息，而是通过特定的 XSL 或者 CSS 来表现，也就是说数据和表现是分离的。而在客户端提交需求后由 XML-enabled Web Server 将内容与形式结合后把最终结果发布给客户端。因此这是一种胖服务器、瘦客户的模式。这种模式同微软将 XSL Parser 集成到 IE 中的设计思想是完全不同的。它的优点是能够对不同设备浏览 Web 提供相应格式的文档，而不需要人工干涉。

从另外一个角度讲，XML Server 可以提供比单纯 XML 文档更强的管理 XML 格式数据的能力，而且可以避免使用传统数据库时需要进行的数据转换过程，（因为 XML 是标准的扩展标记语言，并不是各个公司专属的技术，而数据库厂家对自己的数据库都有不同的格式，所

以必须通过一些中间件进行转换。)而获得高效。

当然,任何技术都不是尽善尽美的,任何一项新的技术在没有完全成熟以前都会存在许多缺点,XML Server 同 XML-enabled Server 也不会例外,它们分别存在如下一些缺点或问题:

对于 XML Server,XML Server 的性能还没有得到验证。因为它采用的是一种全新的数据组织方式,而这种方式在过去还没有得到过广泛的应用。正如我们有理由看好 XML Server 发展前景的同时,我们也有理由对这种没有经过大范围使用的产品持怀疑态度。对于 XML-enabled Web Server 而言:最大的问题是 XML-enabled Web Server 太复杂了,比较起原来用 HTML 和传统 Web Server 建设 Web,懂得使用 XML 这些先进技术的人还是太少了,而且 Server 的安装也显得太复杂,同时开发工具又太专业了。基于 XML 的 Web 服务器体系结构如图 7.5 所示。

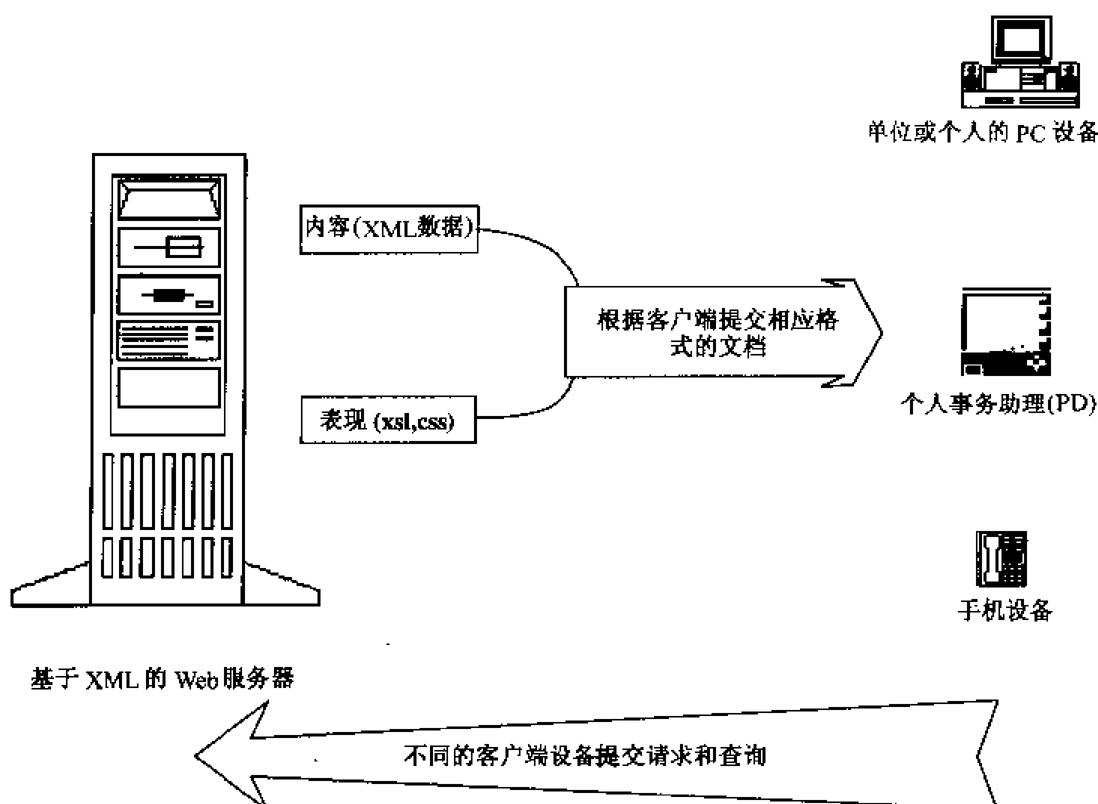


图 7.5 基于 XML 的 Web 信息传送体系结构

最后,我们看一下微软对基于 XML Web 服务器的支持。它的实际实现方法是通过 IIS 的 ISAPI 扩展提供通过 HTTP 直接访问 SQL Server,并将查询结果形成 XML 返回到客户端来完成的。最简单的访问方法是直接在 HTTP 的 URL 中使用 SQL 语句:

```
http://IISServer/VirtualRoot? sql=SELECT+ * +FROM-Customers+FOR+XML+AUTO
```

同时需要注意,我们可以在 URL 中执行存储过程和使用 XML 文档模板。

5. XML 应用服务器

XML 应用服务器实际上是支持 XML 的 Web 应用服务器,它们通常是模板驱动的,通过在一个脚本语言中嵌入使用 SQL 语句,来提取数据并动态构建 XML 文档。

6. 内容管理系统

内容管理系统是用来存储、提取和装配 XML 文档的系统。它们通常包括以下一些特征:

编辑器、版本控制和多用户并发处理，它们自身的数据库实现对用户是透明的，其主要目的是用来管理文档。文档一般以 XML 格式或者其他的如 RTF，PDF，SGML 的形式出现。对于非常简单的文档集合，文件系统就能够满足要求。但是，如果有一个复杂的文档集合的话，通常需要一个内容管理系统。内容管理系统的含义就是允许将文档分割成具体的内容片段，比如例子、过程、章节，或者工具条和其他的一些元数据(如作者名字、版本号等)，然后可以根据需要来重新装配 XML 文档，也可以根据这些片段来合成一个新的 XML 文档。

内容管理系统通常具有下面的一些功能：

(1) 版本和可获取性控制。

(2) 搜索引擎。

(3) 编辑器。

(4) 发布引擎，把内容发布到书本、CD 或者 Web 上。

(5) 内容和形式的分离。

(6) 通过脚本和接口进行扩展。

(7) 和数据库数据集成。

采用对象-关系模型，把 DOM 映射到数据库中，需要为 DOM 中的每一个对象在数据库中建立对应的表格。一般来说，系统需要包括 5 个表：

(1) 属性定义。定义属性，包括它们的类型、合法的值等。

(2) 元素/属性关联。定义哪些属性是和哪些元素相关联的。

(3) 内容模型定义。定义哪些元素能够包含其他的元素。

(4) 属性值。包含属性值和指向在属性定义表和元素/属性关联表中的相关行的指示。

(5) 元素值。包括元素值(PCDATA 或者指向其他元素值的指示)，元素在它的父节点中出现的次数，指向包含父节点元素值所在行的指示，指向在元素/属性表中的相应行的指示。

前三个表和一个简单的 DTD 是等价的，接下来的两个表包含了实际的数据。通过反复的查询后面两个表，就有可能构建 XML 文档的任意一个部分。

7. 可持久化的 DOM 实现(Persistent DOM Implementations)

使用数据库来进行 DOM 实现的考虑是为了加快速度和避免机器内存不够，尤其是在 XML 文档非常大的时候。它们存储了 XML 文档的结构，可持久化的 DOM 实现能够被用来存储、提取和查询 XML 文档，也可以从现在的文档中创建新的文档。也就是说能够通过编程实现基于 DOM 的应用。

实际上，可以在应用程序中，通过编写代码来整合中间件，支持 XML 的数据库、原始 XML 数据库、XML 服务器和永久 DOM 实现等技术。这里 XML 应用服务器需要编写一些脚本代码，内容管理系统需要做一些系统的配置。

7.5 SQL Server 2000 对 XML 存储的支持

Microsoft SQL Server 2000 引入了支持 XML 功能的新特性。这些功能组合在一起使 SQL Server 2000 成为可支持 XML 的数据库服务器。这些新特性包括：

(1) 能够使用 HTTP 访问 SQL Server。

(2) 支持 XDR(XML 数据简化)架构并且能够指定对这些架构的 XPath 查询。

(3) 能够检索并写入 XML 数据。

7.5.1 SQL Server 的 HTTP 访问

(1) 直接在 URL 中指定 SQL 查询,例如:http://IISServer/nwind? sql=SELECT+ * +FROM+Customers + FOR+XML+AUTO&root-root

指定 FOR XML 子句以 XML 文档而不是标准行集的形式返回结果。根参数可标识单一的顶层元素。

(2) 直接在 URL 中指定模板。模板是包含一个或多个 SQL 语句的有效的 XML 文档。模板可以将数据放在一起以形成有效的 XML 文档,但直接在 URL 中指定查询时不一定是这样。例如:

```
http://IISServer/nwind? template = <ROOT + xmlns: sql =" urn: schemas-microsoft-com: xml-sql" >
<sql:query>SELECT+ * +FROM+Customers+FOR+XML+AUTO</sql:query></ROOT>
```

(3) 在 URL 中指定模板文件。在 URL 中写入长 SQL 查询会很麻烦。此外,浏览器对在 URL 中可以输入的文本量可能有限制。若要避免这些问题,可以编写模板并将其存储在文件中。模板是包含一个或多个 SQL 语句和 XPath 查询的有效的 XML 文档。可以在 URL 中直接指定模板文件,例如:

```
http://IISServer/nwind/TemplateVirtualName/templatefile.xml
```

在 URL 中,TemplateVirtualName 是使用用于 SQL Server 的 IIS 虚拟目录管理实用工具创建的 template 类型的虚拟名称。

模板文件还删除来自用户的数据库查询的详细信息以增强安全性。通过将模板文件存储在注册数据库时所在的虚拟根目录(或其子目录)中,删除虚拟根上的 URL 查询处理服务并只允许 SQL Server XML ISAPI 处理文件及返回结果集,从而加强了安全性。

(4) 指定在带批注的 XML 数据简化 (XDR) 架构(也称为映射架构)上执行 XPath 查询。

从概念上讲,对映射架构编写 XPath 查询与使用 CREATE VIEW 语句创建视图并对视图编写 SQL 查询相似,例如:

```
http://IISServer/nwind/SchemaVirtualName/schemafile.xml/Customer[@CustomerID="ALFKI"]
```

在这个 URL 中:SchemaVirtualName 是使用用于 SQL Server 的 IIS 虚拟目录管理实用工具创建的 schema 类型的虚拟名称。Customer[@CustomerID="ALFKI"] 是在该 URL 中指定的 schemafile.xml 上执行的 Xpath 查询。

(5) 直接在 URL 中指定数据库对象。

可以将数据库对象(如表和视图)指定为 URL 的一部分,并对数据库对象指定 Xpath 查询,例如:

```
http://IISServer/nwind/dbobjectVirtualName/XpathQuery
```

在这个 URL 中,dbobjectVirtualName 是使用 SQL Server 的 IIS 虚拟目录管理实用工具创建的 dbobject 类型的虚拟名称。

说明:当在 URL 中执行需要资源(如内存)的操作(创建临时表和临时存储过程、声明游标、执行 sp_xml_preparedocument 等)时,必须执行适当的相应命令(如 DROP TABLE、DROP PROCEDURE、DEALLOCATE 游标或 EXECUTE sp_xml_removedocument)以释放。

图 7.6 显示三层系统构架并描述处理来自客户端的 HTTP 请求的方式。

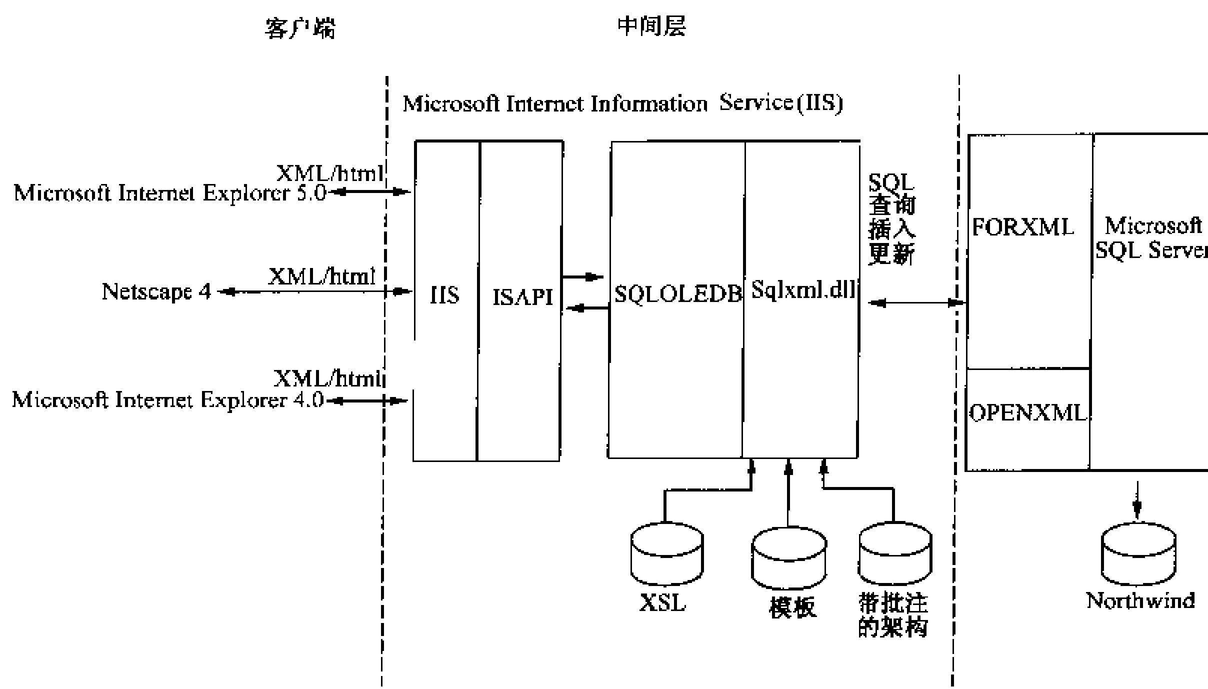


图 7.6 三层系统构架

中间层是 Microsoft Internet Information Service (IIS) 服务器,必须先在该服务器上使用 SQL Server 的 IIS 虚拟目录管理实用工具创建虚拟根。URL 中指定的 IIS 服务器名称用来标识 IIS 服务器。IIS 服务器检查 URL 中指定的虚拟根,并确定是否已注册了 URL 中指定的虚拟根的 ISAPI DLL 扩展 (Sqlisapi. dll)。IIS 服务器装载该 DLL 并将 URL 请求传递到该 DLL。Sqlisapi. dll 扩展与用于 SQL Server 的 OLE DB 提供程序 (SQLOLEDB) 通信,并与虚拟根中标识的 Microsoft SQL Server 实例建立连接。

全部 XML 功能都在 Sqlxml. dll 中执行。当 SQLOLEDB 确定命令是 XML 命令时,该程序便将此命令传递到 Sqlxml. dll,Sqlxml. dll 执行命令并将结果返回给 SQLOLEDB。

模板文件、XML 数据简化 (XDR) 架构文件和可扩展样式表语言 (XSL) 文件都驻留在 IIS 服务器上。XPath 查询和 XDR 架构都在 IIS 服务器上进行处理。Sqlxml. dll 将 XPath 查询转换成 SQL 命令。FOR XML 子句和 OPENXML 都在运行 SQL Server 的服务器上执行。

7.5.2 使用带批注的 XDR 架构创建 XML 视图

可以使用 XDR(简化 XML-Data)架构创建关系数据的 XML 视图。然后可以使用 XPath 查询来查询这些视图。这类似于使用 CREATE VIEW 语句创建视图并指定对视图的 SQL 查询。

XML 架构描述 XML 文档的结构以及对文档中数据的不同约束。当指定对该架构的 XPath 查询时,返回的 XML 文档结构由对其执行 XPath 查询的架构确定。

在 Microsoft SQL Server 2000 中,使用简化 XML-Data (XDR) 语言创建架构。XDR 是一种灵活的语言,它克服了用来描述文档结构的文档类型定义 (DTD) 的一些限制。与 DTD 不同,XDR 架构使用与 XML 文档相同的语法描述文档结构。此外,在 DTD 中,所有数据内容均为字符数据。XDR 语言架构可以指定元素或特性的数据类型。

在 XDR 架构中，<Schema> 元素包含整个架构。作为 <Schema> 元素的属性，可以描述定义架构名称的特性和架构驻留的名字空间。在 XDR 语言中，所有元素声明都必须包含在 <Schema> 元素中。

最小的 XDR 架构如下：

```
<? xml version="1.0" ? >
<Schema xmlns="urn:schemas-microsoft-com:xml-data">
    ...
</Schema>
```

<Schema> 元素是从 xml-data 名字空间（urn:schemas-microsoft-com:xml-data）派生出的。

1.XDR 架构的批注

可以在 XDR 架构中使用对数据库映射的批注来查询该数据库，并以 XML 文档格式返回结果。SQL Server 2000 引入了许多批注，可以使用这些批注将 XDR 架构映射到数据库中的表和列。可以对 XDR 架构所创建的 XML 视图指定 XPath 查询，以查询数据库并获得 XML 格式的结果。

这是替代更复杂的 SQL 查询编写的方法，该方法使用 FOR XML EXPLICIT 模式将 XML 文档结构作为查询的一部分加以描述。有关在 SELECT 查询中使用 FOR XML EXPLICIT 模式的更多信息，请参见使用 EXPLICIT 模式。然而，为克服对映射架构上 XPath 查询的大多数限制，请使用 FOR XML EXPLICIT 模式的 SQL 查询返回 XML 文档格式的结果。

如果拥有公用 XDR 架构(如 Microsoft BizTalk 架构)，可以执行下列任一操作：

编写 FOR XML EXPLICIT 模式的查询，以便生成的数据对公用 XDR 架构有效，然而，编写 FOR XML EXPLICIT 查询可能比较麻烦。

制作公用 XDR 架构的专用复本。然后将批注添加到专用复本，从而生成映射架构。可以指定对映射架构的 XPath 查询。该查询所生成的是公用架构名字空间中的数据。与编写复杂的 FOR XML EXPLICIT 查询相比，创建带批注的架构并指定对该架构的 XPath 查询是一个简单得多的过程。图 7.7 说明了此过程。

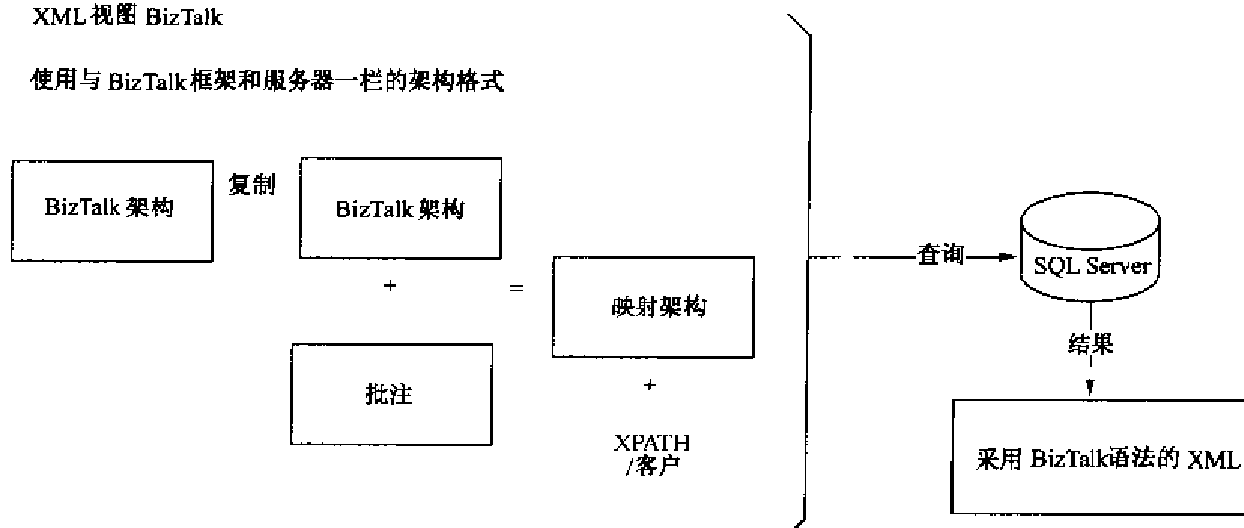


图 7.7　BizTalk 视图

2. 批注的名字空间

XDR 架构中，使用下面的名字空间指定批注：urn：schemas-microsoft-com：xml-sql。

下例显示指定名字空间的最简单方法是在 <Schema> 标记中指定它。urn：schemas-microsoft-com：xml-sql 名字空间的批注必须是由名字空间限定的。

```
<? xml version="1.0" ? >
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql"
              >
  ...........
</Schema>
```

所用的名字空间前缀是任意的。在本文档中，sql 前缀用于表示批注名字空间和使此名字空间中的批注区别于其他名字空间中的批注。

3. 数据类型的名字空间

XDR 架构使您得以指定元素或特性的数据类型。使用下面的名字空间指定数据类型：urn：schemas-microsoft-com：datatypes。

以下是带有名字空间声明的最小 XDR 架构：

```
<? xml version="1.0" ? >
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">
  ...
</Schema>
```

所用的名字空间前缀是任意的。在本文档中，dt 前缀用于表示数据类型名字空间和使此名字空间中的批注区别于其他名字空间中的批注。

<Schema> 元素来源于 xml-data 名字空间：urn：schemas-microsoft-com：xml-data。

4. XDR 架构示例

下例显示如何将批注添加到 XDR 架构中。XDR 架构由 <Employee> 元素和 EmpID、Fname 及 Lname 特性组成。

```
<? xml version="1.0" ? >
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">

<ElementType name="Employee" >
   <AttributeType name="EmpID" />
   <AttributeType name="FName" />
   <AttributeType name="LName" />
   <attribute type="EmpID" />
   <attribute type="FName" />
   <attribute type="LName" />
```

```
</ElementType>
</Schema>
```

现在,将批注添加到此 XDR 架构中,使架构的元素和特性映射到数据库的表和列。带批注的 XDR 架构如下:

```
<? xml version="1.0" ? >
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<ElementType name="Employee" sql:relation="Employees" >
    <AttributeType name="EmpID" />
    <AttributeType name="FName" />
    <AttributeType name="LName" />
    <attribute type="EmpID" sql:field="EmployeeID" />
    <attribute type="FName" sql:field="FirstName" />
    <attribute type="LName" sql:field="LastName" />
</ElementType>
</Schema>
```

在此映射架构中,使用 sql:relation 批注将 <Employee> 元素映射到 Employees 表。使用 sql:field 批注将特性 EmpID、FName 和 LName 映射到 Employees 表中的 EmployeeID、FirstName 和 LastName 列。

此带批注的 XDR 架构提供关系数据的 XML 视图。使用 Xpath 语言可以查询该 XML 视图。Xpath 查询返回 XML 文档形式的结果,而不是 SQL 查询所返回的行集。

7.5.3 检索并写入 XML 数据

微软的 SQL Server 2000 通过三种方式支持 XML 技术:

1.在 SELECT 语句中增加 FOR XML 条件子句

FOR XML 条件子句有三种选择,用来指定如何把 SELECT 语句映射到 XML 上去。RAW 模式指定结果集为一个表格,表格中的每一行对应一个元素,每一列对应元素的属性或者是它包含的子元素。AUTO 和 RAW 的区别在于:行的元素名称和表格名称一致,产生的 XML 文档是线性嵌套的,同表格在 SELECT 语句中出现的顺序相对应。EXPLICIT 允许使用一系列的 SELECT 语句的 UNION 来构建一个 XML 文档。

2.通过 XPath 进行信息定位

通过模式映射的方式,也就是在 XML 文档的元素和属性以及在数据库中的表和字段之间建立映射关系。这里把 XML 看成是一棵对象树,而使用 XPath 的一个子集来进行查询。

3.在存储过程中使用 OpenXML 函数

OpenXML 函数被用来提取 XML 文档的任何一个部分,并把它当做一个表格,然后就可以用 SELECT 语句中的 FROM 指定这个表格,并通过 INSERT 语句在 XML 文档和数据库之间传递数据。并可以用 XPath 来指定具体的元素或者属性。

7.6 Oracle8i 对 XML 存储的支持

Oracle8i 是具有 XML 处理功能的数据库。我们可以利用 Oracle8i 来存储 XML 及建立具

有 XML 功能的应用。把 XML 存储在数据库中,可以使我们从数据库管理工具和过程中受益,例如备份。我们可以使用它们来执行关于数据和安全性的规则,还可以在数据库中嵌入规则和逻辑以阻止破坏数据完整性的操作。同时,把数据库表转换成 XML 文档使我们可以利用 XML 的优势特性。可以利用 XML 文档描述为带有 XSLT 格式页的 HTML 页面,同时可以使用基于 XML 的查询语言进行搜索,同时可以将其作为数据交换格式。

Oracle8i 的对象-关系特性使我们可以获得 XML 数据的复杂结构。我们能够以所希望的粒度来操作和管理 XML 数据,并且可以立即由结果的片断高效地构造出 XML 文档。还可以把 XML 文档与标记作为单独的文档存储在字符大对象(Character Large Object,CLOB)中,或者去除标记后作为数据分布到对象-关系表中。Oracle8i 的 Internet 文件系统可以访问存储在外部文件或 Web 上的 XML 文档。我们可以使用 Oracle8i 的 interMedia Text 搜索引擎在 Oracle8i 所存储的 XML 文档中执行搜索。可以把 XML 作为普通的文字进行索引,也可以把文档分成各节进行更精确的搜索,例如在查找"Oracle WITHWIN title"时"title"就是文档的一节。Oracle8i interMedia 还提供了对文档的全文检索以及执行 SQL 查询能力。

Oracle 提供了一些组件、实用程序和接口,使用户可以在数据库应用中充分利用 XML 技术。这些组件中有 XML 解析器、XSL 转换引擎、XML 类生成器、XML TransViewer 组件以及 XSQL 小型服务程序。XML 解析器是适用于分析 XML 文档的独立 XML 组件。该解析器支持文档对象模型(DOM)和 XML 简易编程接口(SAX)两种接口以及 XML 名字空间。它能以验证和非验证模式分析 XML 文档。在非验证模式中,解析器检验 XML 文档是否是良构的。在验证模式中,它将针对 DTD 验证 XML 文档,检查合法性限制。XSL 转换引擎集成在 XML 解析器中,该转换引擎使用 XSL 格式页来转换 XML 文档。XML 类生成器由 XML DTD 生成 Java 源文件。用户可以用生成的源文件来构造、验证(可选)、打印与输入的 XML DTD 相容的 XML 文档。

XSQL Servlet 是用于处理 SQL 查询的工具,它把结果集作为 XML 输出。该处理程序采用 Java 小型服务程序实现,把包含嵌入式 SQL 查询的 XML 文件作为输入。它可以使用前面的所有组件来执行其操作。XML TransViewer 组件是 XML 组件集合,用于 Java 应用和小应用程序。

7.6.1 Oracle Internet 文件系统(iFS)

Oracle 的 iFS 内嵌了对 XML 文件的支持, iFS 中包含 XML 解析器和转换器,提供了定义新的基于 XML 的文件类型的扩展机制,如图 7.8。

(1) 可以将整个 XML 文档存入一个文本数据块中(BLOB),并序列化为纯文本形式,以关键词的方式来查询 XML 文档。

(2) 将 XML 文档作为数据存储到数据库的表中。支持可扩展的方法来定义文件类型以及解析和转换 XML 文档类型。当注册一个 XML 文档类型时要对数据库提供文档描述符来描述 XML 文档结构类型和在数据库中的存储方式。文件系统的文档描述符是基于 XML 语法的,用来描述基于 XML 的文档

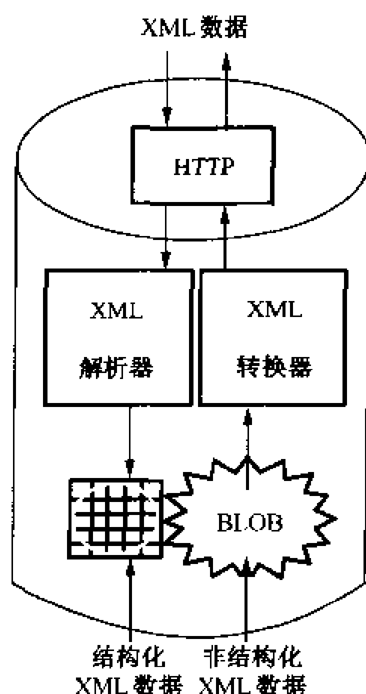


图 7.8 通过数据库视图将 XML 文档结合的示意图

类型结构。XML 文档根据预先注册的文档类型，存入已经在文档描述中描述的表中。

(3) 通过数据库视图将 XML 文档结合起来。Oracle 文件系统允许文档以 XML 的形式提交，也就是说可以将分散在各个表中和文本块中的文档片断，自动组合，产生 XML 文档。利用 Oracle 视图用不同方式获取结构化的和非结构化的 XML 数据，并将其结合为统一的视图。

下例表示一个保险解释文档，其一部分是结构化数据(Payee、Data 等)，一部分是结构化文本标记(DamageReport)。

```
<? xml version="1.0"? >
<InsuranceClaim>
    <ClaimID>12345</ClaimID>
    <LossCategory>7</LossCategory>
    <Settlements>
    <Payment>
    <Payee>Borden Real Estate</Payee>
    <Date>12-OCT-1998</Date>
    <Amount>20000000</Amount>
    <Approver>JCOX</Approver>
    </Payment>
    </Settlements>
    <DamageReport> A massive<Cause>Fire</Cause>ravaged the building and <Casualties>12</
    Casualties>people were killed. Early FBI reports indicate that<Motive>arson</Motive>is suspected.
    </DamageReport>
    ... ... ... .
</InsuranceClaim>
```

7.6.2 XML 基础结构组件之一 —— Oracle XML SQL Utility

通过 XML SQL Utility 也可以操作 XML 数据库。它包括一个 Java 类的集合。功能有：将 XML 文档存入数据库、进行数据库的查询并产生 XML 文档。如图 7.9 所示。

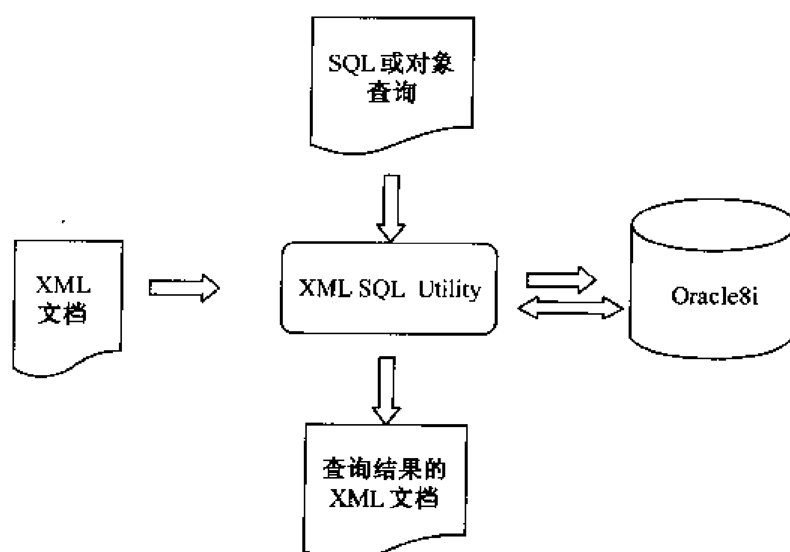


图 7.9 Oracle XML SQL Utility 工作示意

(1) 将 XML 文档写入数据库的表中。

用这种方法将 XML 文档存入数据库中时保留了文档的结构。元素标记名与关系数据库的表中的列名相匹配,有子元素的元素与对象类型相匹配,非结构化的数据如文本注释、描述等不能与数据库的表相匹配,存入数据库的 BLOB 中。下面一段 Java 代码实现了将 emp.xml 中的数据存入数据库的 EMP 表中。这个实例假设 XML 文档结构与数据库的表相一致。

```
import oracle.xml.sql.dml.*;
import java.sql.*;
import oracle.jdbc.driver.*;
import oracle.jdbc.*;
import java.net.*;
public class xmlwritedb
{
  public static void main(String args[]) throws SQLException
  {
    String tabName = "EMP"; // 要插入 XML 数据的表
    String fileName = "emp.xml"; // XML 文档名称
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver()); //初始化 JDBC 连接
    Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:scott/tiger@");
    //将 XML 数据存入数据库表中
    OracleXMLSave save = new OracleXMLSave(conn, tabName);
    URL url = save.createURL(fileName);
    int rowCount = save.insertXML(url);
    System.out.println(" successfully inserted "+rowCount+ " rows into "+ tabName);
    conn.close();
  }
}
```

当然,这要求数据库的表结构与 XML 文档结构一致,反之则需要在存储之前对 XML 文档进行一系列的预处理。比如通过文档分析器将文档分解为文档片断等,将 XML 文档转化成为满足要求的文档。

(2) 通过查询产生 XML 文档。

结果 XML 文档的结构是基于数据库内部视图的。Utility 产生的查询结果可以以字符串来表示,这对于要求返回 XML 文档的请求是比较合适的。也可以通过 XML 的文档对象模型树(DOM)来表示,这适用于程序化的处理 XML 文档(比如用 XSLT 进行数据转换等)。

下面的 Java 代码对数据库进行了查询(select EMPNO,ENAME from EMP),并产生 XML 的结果文档。

```
import java.sql.*;
import java.math.*;
import oracle.xml.sql.query.*;
import oracle.jdbc.*;
import oracle.jdbc.driver.*;
public class xmlquerydb
```

```
{
  public static void main(String args[ ]) throws SQLException
  {
    String tabName = "emp";
    String user = "scott/tiger";
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver()); //初始化 JDBC 连接
    Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:" + user + "@");
    //初始化 OracleXMLQuery  initialize the OracleXMLQuery
    OracleXMLQuery qry = new OracleXMLQuery(conn,"select EMPNO, ENAME from
" + tabName );  //结构化产生的 XML 文档
    qry.setMaxRows(2);  // 设置返回最大的行数值
    qry.setRowsetTag("ROOTDOC"); // 设置文档根节点标记
    qry.setRowTag("DBROW"); // 设置行分离标志
    qry.setStyleSheet("emp.xsl"); // 设置风格表
    //以字符串的形式得到 XML 文档
    String xmlString = qry.getXMLString();
    // 输出 XML 文档
    System.out.println(" OUTPUT IS: \n" + xmlString);
  }
}
```

得到 XML 结果文档：

```
<? xml version="1.0"? >
<ROOTDOC>
  <DBROW id="1">
    <EMPNO>7876</EMPNO>
    <ENAME>ADAMS</ENAME>
  </DBROW>
  <DBROW id="2">
    <EMPNO>7499</EMPNO>
    <ENAME>ALLEN</ENAME>
  </DBROW>
</ROOTDOC>
```

7.6.3 XML 基础结构组件之二 —— Oracle XSQL Servlet

XSQL Servlet 是一个能够处理 SQL 查询并将结果集以 XML 文档的结果输出的工具。该处理程序是一个 Java 小型服务程序，把含有嵌入式 SQL 查询的 XML 文档作为输入。通过它可以很容易地从一个或多个 SQL 查询结果中建立起动态的 XML 数据页。然后通过使用服务器端 XSLT 转换，把其结果用做 Web 上的 XML 数据包或 HTML 页面。SQL Servlet 使用 Oracle XML 解析器、Oracle XSLT 转换引擎以及 Oralce XML SQL 实用程序来完成该工作。

图 7.10 显示了从客户到服务程序又回到客户的数据流情况。事件顺序如下：

(1) 用户在浏览器中输入 URL，它被 Java Web 服务器解释并传送给 XSQL Servlet。URL 中包含目标 XSQL 文件名(.xsql)及可选的参数，例如值以及 XSL 格式页名。用户还可以从命

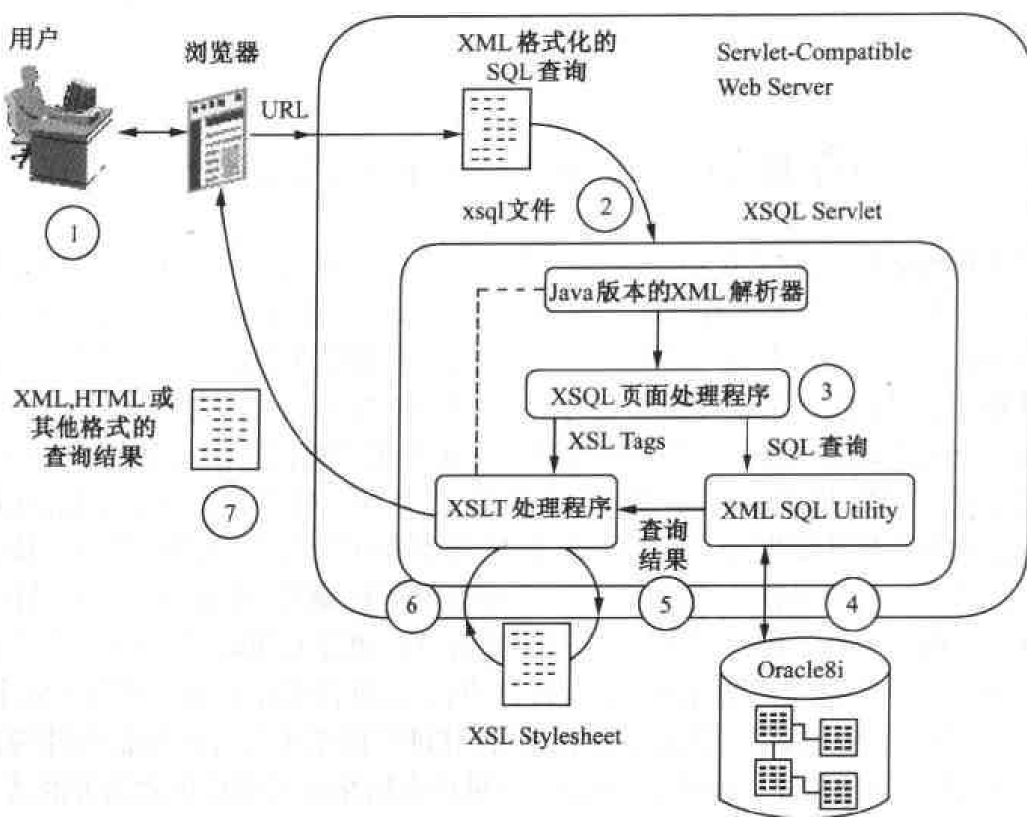


图 7.10　从客户到服务程序又回到客户的数据流情况

令行调用 XSQL Servelet 而绕过浏览器和 Java Web 服务器。

(2) 服务程序把 XSQL 文件传给 Java 版本的 XML 解析器,它会分析 XML 并生成访问 XML 内容的 API。

(3) 服务的页面处理程序组件使用该 API 来获取 XML 参数和 SQL 语句(在<xsql:query>和</xsql:query>之间)。如果有 XSL 处理语句,页面处理程序把它传给 XSLT 处理程序。

(4) SQL Utility 将 SQL 查询传递给潜在的 Oracle8i 数据库。

(5) 返回查询结果,结果被嵌入到 XML 文档原来<xsql:query>标记的同一位置。

(6) 如果需要,查询的结果及其他 XML 数据通过 XSLT 处理程序使用指定的 XSL 格式页进行转换。数据可以转换为 HTML 或格式页所定义的任何其他格式。XSLT 处理程序可以基于发出原来请求的客户的类型而选择性地应用不同的格式页。该信息即 HTTP _ USER _ AGENT,是从客户端通过 HTTP 请求获得的。

(7) XSLT 处理程序把完成后的文档传回客户浏览器,显示给用户。

第 8 章　XML 与 Web 服务

面对今天的网络神话，我们仍然发现存在着巨大的改进空间。今天的因特网在很大程度上还在模仿旧式大型计算机的工作方式。尽管有充足的带宽资源，大量的信息还是被“锁”在了中央数据库里，并由“保安人员”看守。用户必须依靠网络服务器来完成所有的上网操作，这酷似老式的分时复用系统。网站好像孤立的小岛，并不能按照用户的指令在它们之间进行有意义的交流。今天的网络似乎只能通过单个网站向单个用户提供有限的服务——因为大多数的网页只能呈现 HTML 格式的数据“图片”，而非信息本身。对大多数网页来说，在现有技术条件下要两者兼顾是非常困难的。非但如此，浏览器本身在很多方面都只是一个被美化了的“哑巴只读终端”——你可以轻易地浏览信息，但是很难进行编辑、分析和复制（实际上也就是知识工作者需要做的所有工作）。“个性化”只意味着重复地进入网站，并不断地将个人隐私泄露给你所访问的每一个网站。我们必须适应科技，而不是科技反过来适应我们的要求。

微软雄心勃勃的.Net 战略的核心概念就是“把软件当做服务”，也就是把软件应用产品与商业、内容和信息服务合并成一种事物，使之成为可以在网络上订阅使用的服务形式。人们设计、构造、实施、运作、集成和使用软件的方式都将透过网络完成。“总有一天，所有的应用软件将被设计成一种服务，可以在网上订购。”这是微软富有感召力的口号。

Sun 公司也提出了概念相似的开放式网络环境（Open Net Environment，ONE）战略。Oracle 公司则以 Dynamic Web Service 的概念表示与微软抗衡。这些概念的技术取向都是大同小异的，就是都将 XML 作为支持 Web 服务以及各种控制过程的技术核心。

驱动服务的内涵和形式发生变化的是什么？Web 服务模式与以往的服务模式有哪些不同？为什么 XML 会成为 Web 服务的技术核心？XML 如何在 Web 服务中发挥作用？本章将从技术的角度对上述问题进行深入的探讨。

8.1　什么是 Web 服务

一种应用服务代表了某种类型的用户或商务活动，例如阅读电子邮件，接收股市报价、授权赊购业务和购物等。而系统服务代表系统基础设施和管理功能如存储、安全、交易、信息传递和故障克服等。

其实，面向服务的系统并非是个新概念。面向服务的流行系统包括 ONC RPC、DCE、COM、CORBA、RMI 和 Jini 技术。尽管所有这些系统都有许多可圈可点的优势，但他们均需要专用协议实现通信。COM 客户机须使用 COM 协议，与 COM 服务执行通信，而 Jini 客户机须使用 Jini 协议与 Jini 服务实现交互。令人遗憾的是，这些专用协议在 Web 上不流行，而且防火墙往往又为通信设置了障碍。正是在这种背景条件下，一种面向服务的新型系统应运而生，即 Web 服务。

一种 Web 服务代表商务、应用或系统功能的统一，可以通过 Web 接入。Web 服务适用于任何类型的 Web 环境，无论是因特网、内联网还是外联网，重点是企业对消费者、企业对企业、部门对部门或同事对同事的通信。Web 服务消费者可以是通过台式或无线浏览器接入服务

的个人,也可以是应用程序,还可以是另一个 Web 服务。

8.1.1 Web 服务的计算模式

Web 服务彻底地把计算模式从单机、客户机/服务器和 Web 网站的方式转向分布式计算(Distributed Computing)。我们知道,CORBA 和 COM 是今天比较流行的部件对象计算模型。但是它们都存在着“局部计算”的局限性。也就是说,这些模型都仅是本地计算或本网计算的模式,而不能把整个因特网认为是一个计算资源体系来加以利用。Web 服务的技术把分布在因特网上的各种资源有效地通过编程手段整合在特定的应用界面中。Web 服务就相当于过去我们编程中常调用的 API 函数和在面向对象编程中常用的部件接口,只不过 API 一般存在于单个程序的不同模块中,部件接口存在于相同机器的不同部件中,而 Web 服务则将无所不在地分布在网络上。

Web 服务以及各种控制过程都将采纳 XML 的技术作为核心。XML 又被行业称为网络计算的世界语(Lingua Franca),是一种替代 HTML 的可扩展标记语言。XML 的特性决定了其成为 Web 计算模式的基础。

(1) XML 采用文本标记的形式定义各种可交换数据结构,并且可以利用标准的网络协议进行传输。正因为这些特性,XML 实际上代表了平台中性和进行网络计算的趋势。有了 XML 作为核心技术,各种网站提供的服务都不再局限于一些花花绿绿的页面,而是可以进行编程调用的 Web 服务。

(2) XML 是与显示无关的数据表示语言,所以 XML 还支持各种设备和显示环境的自动转换,开发者不必为不同尺寸的设备准备不同的内容和数据。所以 XML 正好合乎微软.Net,SUN ONE 等战略需要透过各种设备访问服务的需求。

(3) XML 是 Web 服务一系列标准的基础。微软正在与包括 IBM 在内的多家公司一起定义围绕 Web 服务的一系列标准,例如 SOAP, WSDL 和 WSML 等。这些标准几乎都是采用 XML 协议簇为基础,保证了未来的软件开发者能够用相同的方式开发调用来自全球的各种服务接口,不管它是一个巨型计算机提供的大数计算服务,还是一个股票交易所提供的行情信息服务。

由于 Web 服务使用 XML 接口和 XML 信息,通过标准 Web 协议执行通信,而且所有的 Web 服务环境都具有互操作特性,至少在理论上是如此。因此,XML 是 Web 服务计算模式的基础。

8.1.2 Web 服务对商业模式的转变

对于消费者来说,Web 服务意味着简单化的整体服务;统一的信息浏览、编辑和授权;查看资料、工作和在线与离线媒体;一种整体的系统方案;随时随地的个性化;绝对的自由。对于个人信息的任何修改——无论是通过个人电脑、便携设备还是灵通卡——将即时和自动地通知到所有需要这些信息的地方。

对于知识工作者和企业,Web 服务意味着统一的信息浏览、编辑和授权;丰富的同步传播;密切的移动通信联系;得力的信息管理和电子商务工具,在基于内部网和因特网的服务程序之间灵活地切换,为动态商务伙伴关系的时代提供支持。

自从 1994 年 Web 问世以来,企业就一直采用因特网密切他们与股东、员工、客户以及伙伴间的关系。他们使用因特网实现供应链自动化,改善商务过程的效益。电子商务再也不是

未来的发展方向，而是成为了一种规范。如今，商家感兴趣的是 Web 服务的因特网计算模式，预期可以实现更优异的跨行业集成，改进效率，密切客户关系。

为使 Web 服务模式适合用户的胃口，商家必须能够利用现有应用资源，并把它们用于 Web 服务。商家需要工具和技术，以便减少转向此种新模式的成本、风险和复杂性。

对于独立软件开发商，Web 服务意味着他们将得到更多的机会，为因特网时代创造更多的新型高级服务——它们可以自动从本地或远程取得和利用所需信息，并通过任何语言和任何设备工作，无须为不同的工作环境重新编写程序。因特网上的一切都变成这种新一代服务程序的潜在构造模块，而每一种应用程序都可以在网上使用。

下面我们以微软的.Net 为例说明 Web 服务对软件商业模式的改变。.Net 为微软带来了一种全新的商业模式。微软预见了"服务"是数字经济的核心商业模式，因此它将逐步转换今天依靠销售盒装软件的获利形式，利用.Net 把"软件作为服务"(Software-As-Service)。微软已经先行一步，把各种软件改造成为能够通过"订阅"方式使用的服务，就像我们今天为使用电话业务、水电和订阅杂志的方式一样，按照"使用量"付费。为此，微软还在.Net 中构造了一系列辅助模块。这些模块包括用户认证、通知和网络存储等功能。一旦微软.Net 的各种基础构造模块开发完毕，微软将最先拿自己的产品进行"服务"尝试。在不远的将来，软件用户将可以体验到在网络上注册使用软件并根据使用时间、存储要求或使用的功能模块多少来付费的方式。此外，微软还会把各种商业服务、内容服务与软件服务并列考虑，这样用户在未来将不会感到"软件"与其他信息产品有任何不同，因为人们真正需要的是快速地达到自己的工作目标，而不是考虑"软件"等产品形式。

未来几年中，微软将逐步尝试把所有的盒装软件产品都改造成为"订阅"使用的方式，微软将突破仅为 PC 设计软件的模式，进而为所有的用户提供各种数字化服务，这将大大提高微软对经济的渗透率，带来更大的商业价值。

8.2 Web 服务的关键技术

当前，已开始出现了一系列支持 Web 服务的技术，处于领先地位的技术包括 UDDI、WSDL、SOAP 和 ebXML。虽然这些技术目前以支持相应软件企业产品的形式出现，但它们正在日臻成熟，逐渐成为事实上的标准。这些技术的共同特点就是以 XML 为核心，规范 Web 服务的描述、服务交互格式和内容，以创造新一代的因特网为目的。

UDDI(Universal Description Discovery and Interoperability：通用描述发现与互操作)是一个由 Accenture、Ariba、Commerce One、康柏、Edifecs、富士、惠普、I2、IBM、英特尔、微软、Oracle、SAP、Sun 和 VeriSign 等公司为首成立的业界联盟组织，现在已有 130 多家公司加入。该组织旨在为 UDDI Business Registry 通用 Web 商务目录，制定相关技术规范。到目前为止，UDDI V1 技术规范已经出台，UDDI Business Registry 也正在执行 Beta 测试。Ariba、IBM 和微软已经开始联合运行 UDDI Business Registry。IBM 已通过其 DeveloperWorks Open Source Zone，发布了适用于 Java 技术开发工具箱的开放型源 UDDI 技术。开发者使用 UDDI 中提供的注册服务发布他们的 Web 服务，并且能够使他们的软件被搜索到为其他开发者提供服务。

WSDL(Web Services Description Language：Web 服务描述语言)是由 Ariba、IBM 和微软共同开发的技术，其宗旨是为描述 Web 服务制定通用的 XML 框架。开发者使用 WSDL 描述他们的服务。WSDL 是一种描述网络服务(Network Service)的 XML 格式，网络服务是能对面

向文档类型的信息和面向过程的信息进行操作的端点(Endpoint)的集合。对操作和消息的描述是抽象性的,并在定义端点时,将消息和操作绑定到具体的网络协议和消息格式上。相关具体的端点集构成抽象的端点集(服务)。WSDL是可扩展的,它允许对端点和端点间的消息进行描述,同时不去考虑具体的消息格式或者双方用于通信的网络协议。迄今为止,IBM已通过其alphaWorks发布了面向Java技术开发工具箱的WSDL技术。

SOAP(Simple Object Access Protocol:简单对象访问协议)是由DevelopMentor、IBM、Lotus、微软和Userland等公司开发的技术。SOAP提供了一种可扩展XML信息传递协议,并支持RPC编程模式。许多SOAP的执行方案已经面世,最受青睐的两个方案是Apache软件基金会的开放式源Java技术执行方案和微软在.NET SDK的执行方案。尽管两个方案之间的互操作问题依然困扰着开发人员,但其性能还是相当稳定的。

惠普和微软已开发了SOAP的扩展型技术,名为SOAP Messages with Attachments。微软的BizTalk Server 2000采用了SOAP Messages with Attachments,因此该项技术包含在一个支持性产品中。最近W3C(World Wide Web联盟)成立了一个XML协议工作组,目的是开发标准的XML信息传递协议(XP)。SOAP开发人员已向W3C递交了SOAP技术规范,他们把该规范作为XP项目的一个起点。SOAP Messages with Attachments也递交给了W3C。但W3C尚未向外界透露其工作进展的细节。开发者使用SOAP将他们的应用与Web服务进行绑定,调用服务。

ebXML(电子商务XML)是由ebXML Initiative开发的一种B2B XML框架。ebXML Initiative是UN/CEFACT(联合国贸易促进与电子商务组织)和OASIS(结构化信息标准促进组织)执行的联合项目。ebXML成员包括来自2000多个公司、政府部门、科研团体、标准组织的代表和来自全球各地的个人。

ebXML是一种全面的B2B框架,通过共享基于Web的商务服务,实现企业协作。该框架支持B2B商务过程的定义与执行,B2B商务过程以商务服务交换的设计顺序来表示。该框架包括信息服务、协作伙伴协议、核心部件、商务过程方法、注册器与仓库等技术规范。2000年10月,执行了一个互操作方案的验证演示,参加的成员包括Ajuba Solutions、Cisco、Extol、Fujitsu、IBM、IPNet、Netfish、NTT、Savvion、Sterling Commerce、Sun、TIE、Viquity、WebMethods、XML Global和XML Solutions。

8.3 Web服务的体系结构

到目前为止,并没有一个完整的Web服务体系结构参考标准。Web服务体系结构主要是根据不同软件企业的Web服务战略而制定的。通过这些来自于不同企业的Web服务体系结构,我们能够发现Web服务的基本组成要素。在本节,我们重点介绍具有代表性的Sun ONE软件体系结构和微软的.NET体系结构。微软的.NET战略是一个长期的非常宏大的计划,因此它的体系结构更具有方向性和全局性,从而具有更高的抽象层次。Sun ONE的技术原则是采用通用和流行技术,也就是说使用现行可用的技术,而不是要重新发明。因此Sun ONE的体系结构更加具体,也更加清晰。

我们希望通过介绍两个不同层次的Web服务体系结构,使读者对Web服务有一个全面的认识。

8.3.1 Sun ONE 软件体系结构

Sun ONE 软件体系结构是一种生机勃勃的系统体系，随着有利于增强该软件环境的新技术的不断出现，它将继续扩展。从本质上讲，Sun ONE 软件构架基于 XML、Java 技术和 LDAP。Sun ONE 构架使用的核心标准如下：

(1) XML 标准与技术，其中包括核心 W3C XML 建议（XML、DOM、XML Namespaces、XSLT、XPath、XLink 和 XPointer）；开发团体的 XML 分析器、SAX；表现格式（XHTML、WML 和 VoiceXML）；XML 模式系统（DTD、XML 模式、RELAX 和正在出现的 TREX 技术规范）；XML 信息传递系统（新出现的 W3C XML 协议、SOAP 和 ebXML 信息服务）；XML 注册器与仓库（UDDI Business Registry、ebXML 注册器与仓库以及 OASIS xml.org）；XML 服务描述语言（WSDL）；XML 管理信息交换框架（DMTF CIM 与 WBTM）；B2B XML 框架（ebXML 协作伙伴协议、ebXML 商务过程方法和 ebXML 核心组件）；XML 安全系统（XML 签名、XML 加密以及新出现的 OASIS 安全服务项目）。

(2) Java 技术，其中包括 Java 2 平台企业版（J2EETM）、Java 2 平台标准版（J2SETM）、Java 2 平台 Micro 版（J2METM）、MID（Mobile Information Device）规定、Java CardTM API、Java Servcet 与 JavaServer PagesTM（JSPTM）软件、适用于数据库访问的 Java API（JDBC）、适用于 XML 的 Java API（适用于 XML 处理的 Java API、适用于 XML 数据联编的 Java API、适用于 XML 注册的 Java API、适用于 XML 信息传递的 Java API）。

(3) 基础设施标准，例如 HTTP、SLL 和 LDAP。

图 8.1 概括显示了 Web 服务体系结构主要部件的功能。

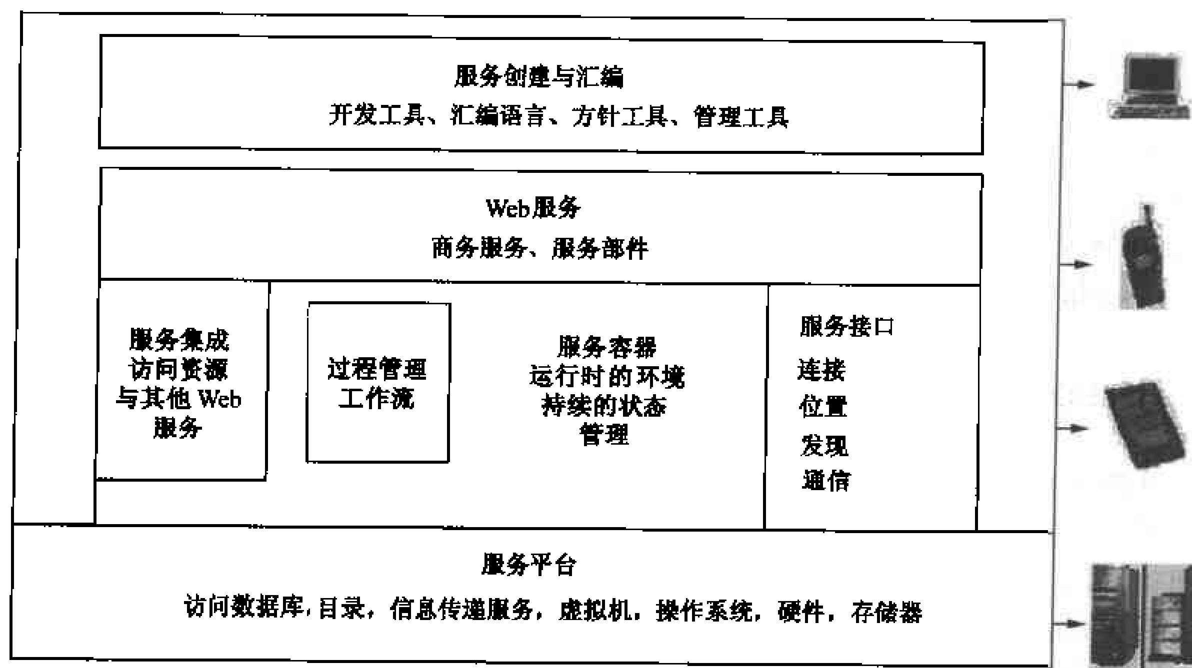


图 8.1 SUN ONE Web 服务体系结构主要部件功能概况

图 8.1 顶部是服务创建与汇编框。这一部分显示根据 Web 服务模式开发系统而使用的工具。通常，Web 服务由一系列分离式服务部件组成，这些部件采用实例描述各种内容、商务逻辑和应用。因此，服务开发过程涉及两个不同阶段。

第一阶段涉及创建分离式服务，Sun ONE 软件体系结构称之为微观服务。第二阶段是把微观服务汇编成整合服务或宏观服务。开发人员使用集成开发环境、代码生成器、XML 编辑器和创作工具，构建微观服务。服务汇编人员使用商用过程模式工具、工作流工具和其他类型的粘贴工具，汇编宏观服务。汇编人员还使用方针工具定义商务规则、安全方针以及运行时可以动态方式改变宏观服务的与用户信息相关的方针。服务测试完毕后，就可以随时在部署平台安装，方针和规则可以部署于一个开放式目录。管理工具提供部署和管理服务环境的功能。

位于图 8.1 服务创建与汇编框下部的是已部署服务框，它由商务服务（宏观服务）与服务部件（微观服务）组成。微观服务可以汇编、配置或再配置成任何数量的宏观服务。

图 8.1 的其余部分显示 Sun ONE 软件部署体系结构。图形从右至左显示服务消费者与服务接口的交互情况。服务接口提供连接、位置、发现和通信等基本功能。体系结构的这一部分包括各类用户和允许 Web 用户、无线用户、语音用户以及外部商务系统调用服务的各类商务门户。服务接口把每个请求定向于相关的 Web 服务。

微观服务和宏观服务在某种类型的服务容器内执行。服务容器为服务提供运行时的环境，并为服务提供持续的状态管理。服务容器运行于服务平台，提供对数据库、目录和信息传递服务的访问。服务平台驻留于操作系统或虚拟机，提供对硬件、存储器和网络的访问。值得注意的是，Web 服务可以部署于任何类型的平台或智能化设备。

过程管理工具与服务容器进行协同工作，管理服务工作流和事件处理。服务集成工具使用基础服务平台，访问其他 Web 服务与资源，例如数据库、文件、记录和传统应用。

8.3.2 微软.NET 框架

建立在 XML 和因特网协议的标准整合构造上的微软.NET 平台为开发新型高级软件提供了一个革命性的模型。在此之前，软件开发模式一向侧重于单个系统，甚至试图掩饰与其他系统的互动，使它们看起来像本地系统内部的互动。微软.NET 的设计意图于分明确，要将网上所有可用资源整合成为一个解决方案，而对现有的技术来说，这种整合极其复杂和昂贵的。微软.NET 将使这一点成为所有软件开发活动的内在本质。

被称为“Web 服务引擎”的.NET 框架需要两种基本结构作为它的基础。

8.3.2.1 松散耦合

.NET 需要通过将紧密耦合的、高效的 n 层计算技术与面向消息的、松散耦合的 Web 概念相结合。Web 服务是一种应用程序，它可以使用标准的因特网协议，像超文本传输协议（HTTP）和 XML，将功能纲领性地体现在因特网和企业内部网上。可将 Web 服务视作 Web 上的组件编程。

从理论上讲，开发人员可通过调用 Web 应用编程接口（API）（就像调用本地服务一样），将 Web 服务集成到应用程序中，不同的是 Web API 调用可通过因特网发送给位于远程系统中的某一服务。例如，Microsoft Passport 服务使得开发人员能够对某应用程序进行验证。通过 Passport 服务编程，开发人员可以充分利用 Passport 的基本结构，通过运行 Passport 来维护用户数据库，以确保它的正常运行、定期备份等。

在某个网络中分发应用程序逻辑，并不是一个全新的概念，在 Web 中分发并集成应用程序逻辑才是一个崭新的概念。

从前，分布式的应用程序逻辑需要使用分布式的对象模型，如微软的分布式组件对象模型(DCOM)、对象管理集团的公用对象请求代理程序体系结构(CORBA)或 Sun 的远程方法调用(RMI)。通过使用这种基本结构，开发人员仍可拥有使用本地模型所提供的丰富资源和精确性，并可将服务置于远程系统中。

这些系统有一个共同的缺陷，那就是它们无法扩展到因特网上：它们要求服务客户端与系统提供的服务本身之间必须进行紧密耦合，即要求一个同类基本结构。这样的系统往往十分脆弱：如果一端的执行机制发生变化，那么另一端便会崩溃。如果服务器应用程序的接口发生更改，那么客户端便会崩溃。

提供紧密耦合的基本结构也很常见，许多应用程序均是基于这种系统构建而成的。但是，当各个公司需要相互合作或信息技术提供商扩大业务范围时，便很难实现单一而统一的基本结构。根本无法保证希望与之进行远程通信的管道的另一端，具备所有需要的基本结构、对于它使用的操作系统、对象模型或编程语言，可能一无所知。

相反，Web 服务彼此是松散耦合的。连接中的任何一方均可更改执行机制，却不影响应用程序的正常运行。从技术角度讲，人们已转向使用一种基于消息的异步技术来实现高可靠性的系统性能，通过使用诸如 HTTP、简单邮件传输协议(SMTP)以及至为重要的 XML 来实现统一的连接。

消息传递系统将通信的基本单元打包成自我描述型的数据包(又称做消息)，并将其放到网络缆线中。消息传递系统与分布式对象系统之间的本质区别在于：要求发送方辨识接收方的基本结构的程度有所不同。在分布式系统中，发送方需对接收方的情况进行种种猜测：应用程序是如何激活或拆包的，调用的是什么样的界面等。

另一方面，消息传递系统会在缆线格式级上创建合同。发送方既不需考虑消息被接收后的情况，也不需考虑接发双方之间的通信情况，惟一需要考虑的是接收方是否能辨识发送的消息内容。

在缆线格式级上创建合同的优势不言而喻。例如，接收方可在任何时刻进行更改，而不会干扰发送方的消息发送，只要它仍可辨识原有消息的内容。另外，发送方无需任何特殊的软件即可与接收方通信：只要它发出正确格式的消息，接收方就可以响应。

8.3.2.2 缆线级的 XML:SOAP

实现 Web 服务的基本结构以及在整个 Web 中实现 Web 服务的关键是实现支持简单数据描述格式的技术，这种格式就是 XML。Web 服务必须使用 XML 来完成三件事情：基本的缆线格式、服务描述以及“服务发现”。

(1) SOAP：在通信的最低级别，系统需要使用同一语言。作为通信双方的应用程序需要遵守同一套通信规则：如何表示不同的数据类型(例如：是整数还是数组)以及如何表示命令(即需要对数据进行何种操作)。另外，在必要的时候应用程序还需对该语言适当的扩展。简单对象访问协议(SOAP)是 XML 的实施工具，它提供了一套公共规则集，该规则集说明了如何表示并扩展数据和命令。

(2) Web 服务描述语言(WSDL)。双方应用程序在得到了如何表示数据类型和命令的规则后，需要对所接收的特定数据和命令进行有效的描述。在接收到两个整数后，应用程序必须明确表述它可以对这两个整数执行乘法运算操作。Web 服务描述语言(WSDL)是一种 XML 语法，开发人员和开发工具可使用它来表述 Web 服务的具体功能。

(3)“SOAP 发现”。在最高层,还需制定一套如何定位服务描述的规则:默认情况下,用户或工具能在什么地方找到服务的功能描述?依据“SOAP 发现”规格说明中提供的规则集,用户或开发工具可以自动找到服务的 SCL 描述。

一旦实现了这三种功能层,开发人员便可容易地找到 Web 服务,将它例示成一个对象后再集成进应用程序中,继而构建出一个具有丰富功能的基本结构。这样得到的应用程序便能与 Web 服务进行反向通信了。

8.3.2.3 .NET 框架的组成

很显然,许多基本结构都需实现上述进程对开发人员和用户的透明化。.NET 框架提供此基本结构。从.NET 框架角度看,所有组件都可以是 Web 服务,而 Web 服务也仅是一种组件。实际上,.NET 框架提取出微软组件对象模型(COM)的精华,将它们与松散耦合计算的精华有机地结合在一起,生成了强大、高效的 Web 组件系统,简化程序员的“管道”操作、深入地集成了安全性,引进了基于互联网的操作系统,极大地改善应用程序的可靠性和可扩展性。

.NET 框架包含三个主要部分:公共语言运行时(Common Language Runtime,CLR)、具有多层次结构的统一的类库集合和高级版“活动服务器页面”(又名 ASP+)。

公共语言运行时

此名称不能准确反映它的全部功能。实际上,公共语言运行时在组件的开发过程中以及软件的运行过程中,都扮演着非常重要的角色。在组件运行过程中,运行时负责管理内存分配、启动或取消线程和进程、实施安全性策略,同时满足当前组件对其他组件的需求。在开发阶段,运行时的作用有些变化:与现今的 COM 相比,运行时的自动化程度大为提高(比如可自动执行内存管理),因而开发人员的工作变得非常轻松。尤其是映射功能将使代码编写量锐减,这些代码是开发人员在将业务逻辑转化成可复用的组件进行编程时所需的。

对编程语言而言,运行时这个概念并不新奇:实际上每种编程语言都有自己的运行时。Visual Basic 开发系统具有最为明显的运行时(名为 VBRUN),Visual C++ 跟 Visual FoxPro、JScript、SmallTalk、Perl、Python 和 Java 一样,有一个运行时 MSVCRT。.NET 框架的关键作用是它提供了一个跨编程语言的统一的编程环境,这也是它能独树一帜的根本原因所在。

统一的编程类

.NET 框架中的类为开发人员提供了一个统一的、面向对象的、层次化的、可扩展的类库集(API)。现今,C++ 开发人员使用的是微软基础类库,Java 开发人员使用的是 Windows 基础类库,而 Visual Basic 用户使用的又是 Visual Basic API 集。简而言之,.NET 框架统一了微软当前各种不同的框架。这样,开发人员不再需要学习多种框架就能顺利编程。除此之外,通过创建一个公共的跨编程语言的 API 集,.NET 框架可实现跨语言继承性、错误处理功能和调试功能。实际上,从 JScript 到 C++ 的所有编程语言,都是相互等同的,开发人员可以自由选择理想的编程语言。

高级版“活动服务器页面”(ASP+)

ASP+ 是使用 .NET 框架提供的类库构建而成的,它提供了一个 Web 应用程序模型,该模型由一组控件和一个基本结构组成。有了它,Web 应用程序的构建变得非常容易。开发人员可以直接使用 ASP+ 控件集,该控件集封装了公共的、用于超文本标识语言(HTML)用户界面的各种小组件(诸如文本框、下拉菜单等)。实际上,这些控件运行在 Web 服务器上,它们将用户界面转换成 HTML 格式后再发送给浏览器。在服务器上,控件负责将面向对象的编程模

型呈现给 Web 开发人员，这种编程模型能提供面向对象的编程技术拥有的丰富功能。ASP+还提供一些基本结构服务(诸如会话状态管理和进程循环)，这些服务进一步减少了开发人员要编写的代码量，并使应用程序的可靠性得到了大幅度提高。ASP+还允许开发人员将软件作为一项服务进行传送。通过使用 ASP+ Web 服务功能，ASP+开发人员只需进行简单的业务逻辑编程，而由 ASP+基本结构负责通过 SOAP 传送服务。

尽管 ASP+还未正式发行，但它已在改进应用程序功能方面创造出令人难以置信的奇迹：在现有基于 ASP 的应用程序性能基础上，性能优化了三倍之多，更为激动人心的是生产效率再度攀升。

.NET 框架的核心要素

.NET 框架有几个要素比较重要。首先是它的安全系统和配置系统。这两个系统协同工作，有力地遏止了运行不安全代码的可能性，并大幅度减少了号称"DLL Hell"的对应用程序进行配置时所面临的挑战。

安全系统是一个高度细化、基于事实的系统，它赋予开发人员和管理员多种代码处理权限(而不仅仅是"on"或"off")。将来还会根据代码本身的核心要素来决定如何实施上述权限。

当.NET 框架应用程序被下载到某一系统中时，它会申请一组权限(诸如对临时目录的写入权限)。运行时将收集有关应用程序的事实信息(诸如它是从何处下载的，是否用了有效签名，甚至它访问系统的准确程度)，并按管理策略决定是否允许应用程序运行。运行时甚至还可告知应用程序它无法授权申请的所有权限，并允许应用程序自行决定是否继续运行。

有这种安全系统作保障，许多应用程序配置问题便会迎刃而解。开发人员和管理员(最终是用户)所面临的最大挑战之一是版本的管理问题。如果在新装了某个应用程序之后，一切都限于瘫痪状态，而在这之前系统一直运行得非常良好，那么最大的可能是新安装的应用程序重写了一些共享库，并极有可能修正了现有应用程序正使用的程序错误。这种情况出现的频率很高，以致人们将它称为："DLL Hell"。

.NET 框架拥有的几项高级功能可以彻底消除"DLL Hell"现象。首先，它有一个非常强大的内部命名系统，能够有效地防止两个库因互相重名而被错当为对方的情况发生。除此之外，它还提供一项被称为"并行"配置的新功能。如果前例中新安装的应用程序确实重写了共享库，现有应用程序可对该库进行修复。现有应用程序再次启动时，它会检查所有的共享文件。如果发现文件被更改，同时这些更改又是不兼容的，则它可以请求运行时提取一个可以使用的版本。得益于强大的安全系统，运行时可以安全地执行该操作，这样应用程序就完成了本身的修复工作。

8.4 智能化 Web 服务

从 Web 出现的第一天起，让 Web 更加智能的努力从来没有停止过。但是到目前为止，计算技术仍然被分割在两个领域里——个人电脑和相关设备的应用程序领域和网站领域。人们希望可以让这两个紧密地合作，将个人电脑的强大功能与因特网上无尽的信息海洋结合起来，希望把今天的网络变成Tim Berners-Lee(WWW 的发明者)曾经预言的"互动式创新空间"。

8.4.1 对智能化 Web 服务的理解

从现有和可预期技术的角度出发,人们希望 Web 服务能够比现有网络更加智能。人们对智能的理解千差万别,Web 服务能够在哪些方面提供更加智能的服务?

首先是更加方便快捷的工作方式。今天,工作方式更多的是分散而非整体的。网络浏览(只读)、创意(写作与编辑)、沟通(电子邮件和即时讯息)、日程安排和联系(离线,依靠设备),它们所需的应用软件在种类、功能和通配性方面相去甚远。多数人倾向于使用单一而全面、能够适应任何工作性质的操作环境,明白无误地切换于本地和远程服务商的应用程序之间,并且基本上依靠设备自身工作。微软.NET 战略将其 Web 服务的智能化加强描述为一系列产品支持,它们包括:

自然界面——新一代"人-机交互技术的总成",包括通过"键入盒"把语音、视觉、手写和自然语言录入。这些技术可以针对多模式用户界面进行重组。自然界面为所有的设备和操作环境提供了常规的用户技术。

通用画布——这是一个 XML 复合信息构架,将浏览、通信和资料撰写集成在统一而全面的环境中,使用户能够以统一方式进行信息综合与互动。建立在 XML 规划基础上的通用画布把因特网从只读环境转换成读写平台,让用户以互动的方式撰写、浏览、编辑、注释和分析信息。因为所有信息的表述基础都是 XML 格式,所以通用画布可以将来自世界各个角落的信息聚集在一起,供用户阅读和综合利用。

信息代理——帮助管理用户在因特网上的身份和角色,令用户更好地控制网络和服务商以何种方式互动。它同时维护历史记录、设备环境和选择的优先顺序,也就是因特网生活的过去、现在和未来。它支持例如 P3P 等隐私保证技术。与今天的因特网不同,个人信息始终掌握在自己的手中,由自己来决定让谁访问。

智能标记——将 IntelliSense 技术延伸到网络内容上,使个人电脑和相关设备聪明地处理来自网上的信息。这种可扩展的构架让任何人都能够创造适应性用户技巧和数据帮手。它是 XML 规划的内在能力。

8.4.2 智能化 Web 服务中的用户信息共享问题

我们可以看出,智能 Web 服务的核心是以人为本,紧密围绕用户。其中,一个很重要的问题是需要共享用户信息。所谓的用户信息是指 Web 服务需要了解有关服务消费者的情况,以便提供定制化和个性化的服务。用户信息指消费者身份、地址以及与消费者信息相关的隐私权限制。如要把许多服务汇编成一个整合型商务服务,这些服务就需要共享用户的这些信息。

以共享用户身份为例。提供拥有个性化特点的 Web 服务,将维护每个用户身份的信息。当个人使用服务时,他要向服务自报身份。有时,Web 服务将根据个人系统中保存的一个 Cookie 自动确定他的身份。有时,用户必须输入用户 ID 和口令。

任何人都能够创建与用户信息相关的 Web 服务。现在就有许多 Web 网站,根据用户访问网站的历史记录,提供高度定制化的服务。但符合用户的具体情况还只是问题的一半。智能化服务还必须能够和其他服务共享用户信息。令人遗憾的是,目前缺少代表这些用户信息的标准和规范。提供个性化服务的每个网站,都采用专用格式保存用户身份和历史情况等信息。

在对广泛的分布式异构 Web 服务有透彻的了解和实现动态的互动之前,首先必须解决共

享用户信息的问题，但靠一家供应商是无力解决这个问题的。解决方案不能是专用的，而必须是开放的和可以互操作的，必须与任何其他的 Web 服务执行协作。同时，解决方案必须保护个人信息的安全与隐私权。

智能化 Web 服务是指能够分析具体的用户信息，并能够与其他服务共享用户信息的 Web 服务。它能够根据“谁”、“什么”、“何时”、“何地”和“为什么”诸要素，生成动态结果。智能化 Web 服务可以识别大量的用户信息，这主要包括：

(1) 服务的消费者的身份，是个人还是公司。

(2) 消费者在调用服务时使用的职务。

(3) 消费者可能为某种服务定义的优先选择。

(4) 与消费者相关的安全方针。

(5) 与消费者相关的隐私权方针。

(6) 与消费者相关的商务方针。

(7) 消费者的具体的地理位置。

(8) 调用服务时使用的客户机设备类型。

(9) 与本服务或相关服务消费者相关的以往的历史情况。

(10) 消费者与本服务供应商之间存在的任何服务级别协议。

下面举一个直观的有关个人信息服务的例子。例如，调用智能化饭店查找服务，选一家饭店。该项智能服务需要根据你住在何处，你喜欢什么，你最近去了哪家饭店，你要和谁一块进餐等情况，提出一个建议。你是和自己的孩子还是陪一位重要的客户共同进餐，情况可大不一样。因此，智能化饭店查找服务需要考虑你的身份、爱好、历史和职务等。

在电子商务领域，我们以订单执行过程为例。设想一个客户订购了六种不同的产品。客户下订单时，你承诺在两日内发货。但是，当采购服务部门向合同供货机构下订单时，却发现有一种产品供货受限，使你无法按预先的承诺向客户发送订购的所有产品。在诸如此类的情况下，你的标准政策是发运部分产品，但这样做将使利润受损。即使执行了多次发货，但产品却往往完全是同日到货，尽管客户或许不介意是今日还是明日到货，但是通常引起不快。因此，需求一种既节省成本，又能使客户满意的两全其美的方案。

智能 Web 服务不会自动执行多次发运，而是首先考虑客户概情况和过去历史，权衡赶制缺货部件与调整工厂生产计划所产生的利润影响，然后再据此确定一系列行动。智能 Web 服务还可能考虑库存成本、空间成本、预计运费和预计交付时间等其他参数。它可能采用电子邮件、SMS 信息、自动语音应答信息、回复传真或客户概况中给出的其他某种交付渠道，向客户发送如下的查询，并请求客户作出决定。

1. 鉴于给你带来的不便，请接受我们提供的这项优惠券。一旦订单执行完毕，我们将即刻发货，目前估计发送需要 5 天时间。

2. 你订购的产品一旦有货，将分批发送。

8.4.3 为 Web 服务增加智能

人们需要的是代表用户信息的一套新标准和一种 XML 框架，另外还需要一种开放式体系结构，以便定义智能化服务如何使用用户信息，保障服务的操作性能。

以 SUN ONE 的体系结构为例，开放式的智能化 Web 服务体系结构对 Web 服务体系结构进行了增强(图 8.2)。智能化方针的功能是根据由用户身份、相关信息和职务等有关方针来

协调各种相应的活动。智能化交付工具可根据用户信息,集合服务结构并实现服务结果的定制化与个性化。智能化过程是根据用户信息,去更改商务服务工作流的过程。智能化管理将根据用户信息所确定的具体情况,保障用户的隐私权、安全性和访问权力。

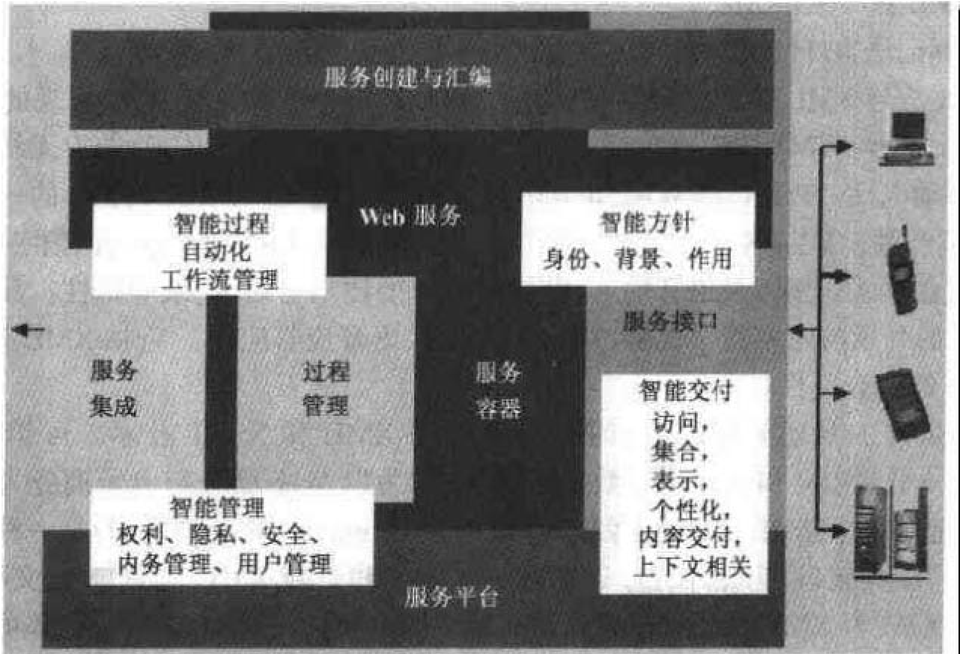


图 8.2 提升 Web 服务智能化水平

8.4.3.1 提高智能化的协议保证

图 8.3 显示了支持智能化 Web 服务体系结构各组成部分的标准与技术。这些标准和技术为 Sun ONE 软件体系结构奠定了基础。并非每一种情况均需要所有这些标准,它们是作为实现互操作的指导原则和辅助手段而提供的。同时,由于这些标注和技术均是采用通用和流行的,因此具有一定的普遍意义和指导意义。



图 8.3 支持智能化 Web 服务的标准

8.4.3.2 智能化交付

智能化交付支持各种各样使用一系列设备专用表示格式的客户机，其中包括 HTML、XHTML、WML 和 VoiceXML。

超文本标记语言(HTML)是向用户交付电子信息最常用的技术。

XHTML 是与 XML 兼容的变体 HTML。HTML 和 XHTML 是 W3C 实现的技术。

无线标记语言(WML:Wireless Markup Language)是用于众多 WAP 兼容无线设备的表示格式，例如移动电话。WML 是 WAP 论坛开发的技术，当前已采用 XHTML 的一个子集作为 WML 的后续产品。VoiceXML 允许创建声频对话，声频对话拥有合成语音、数字化声频、语音和 HTMF 键盘输入识别以及语音输入、电话与混合型技术会话、记录等特性。其主要目标是把基于 Web 的开发与内容交付优势，融合于交互式语音应答应用。VoiceXML 是 VoiceXML 论坛与 W3C 的联合项目。

智能化交付工具还管理内容转换，通常使用 XML 和 XSLT 技术。可扩展标记语言(XML)是一种平台中立和语音中立数据表示格式，为以 Web 服务与电子商务为重心的每个重要项目打造了一个基础。文档对象模式(DOM:Document Object Model)是一种平台中立和语言中立应用编程接口，它允许程序以动态方式访问和更新 XML 与 HTML 文档的内容、结构及格式。DOM 为 XML 数据提供存储器树形结构访问，这样整个文档结构都可以执行读和写访问。Simple API for XML (SAX)提供基于事件的串行 XML 分析程序。Extensible Stylesheet Language Transformations (XSLT) 是把 XML 文档转换成 XHTML、WML、VoiceXML、其他 XML 文档或其他数据格式的编程语言。XML、DOM 和 XSLT 是 W3C 的工作项目。SAX 是 XML-DEV 讨论组成员的一个协作项目。

Web 服务显示一个 XML 接口，它是使用 XML 信息传递协议予以实现的，例如 SOAP、ebXML 信息服务(ebXML MS)或新出现的 W3C XP。诸如文档类型定义(DTD)、XML Schema 和 RELAX 等 XML 模式语言或新出现的 TREX 技术规范，旨在描述 XML 信息的格式。信息模式在一个模式注册器里执行注册，例如 xml.org。Web 服务可使用 WSDL 等 Web 服务描述语言执行描述，在 UDDI Business Registry 和/或 ebXML Registry 与 Repository 执行注册。

8.4.3.3 服务容器

服务容器为 Web 服务提供运行时的环境。使用的服务容器类型，取决于托管服务所使用的平台类型。Sun ONE 软件体系结构为相关服务器和其他设备都提出了很好的建议。

基于服务器的被建议的服务容器是 J2EE 应用服务器。基于 J2EE 技术的服务，可以作为服务器小程序、JSP 页面或 Enterprise JavaBeans (EJB)部件予以实现。当前，J2EE 应用服务器以透明方式，管理与 Web 服务相关的大量复杂部署与执行。当 Web 应请求提供较多的用户信息以及有关服务的可用信息时，J2EE 容器将发挥极其重要的作用，即以透明方式利用这些信息配置和定制 Web 服务。J2EE 容器将扩展其现行说明方式，以使智能服务实现自动化开发与部署，不需要开发人员执行复杂的编程任务。这种说明方式现已应用于资源管理、交易管理、状态管理和安全等领域。J2EE 说明编程技术大大减少了 Web 服务开发的复杂性。

基于设备被建议的服务容器是 J2ME 技术，它为基于无线电话、PDA、电视、汽车信息系统和其他多种消费者设备与嵌入式设备的 Web 服务提供一个部署平台。J2ME 设备在一个启动文件里执行定义，其中包括 Java 虚拟机和一系列 API，它们提供访问基础设备的能力。启动

文件提供必要的抽象过程，允许跨越极其广泛、种类繁多的网络和设备，以透明方式部署服务与内容。当前可用的技术包括适用于无线电话和PDA的移动信息设备启动文件(MIDP)与适用于广播电视、机顶盒和个人录相机的Java TV API。这些平台提供的应用模式，使服务可以访问设备的许多特性，包括存储器、用户界面和个性化服务等。

8.4.3.4 智能化管理

从传统观念看，系统与网络管理一直是使用大型分布式应用予以实现的。在这种情况下，系统管理软件供应商的专用代理软件，安装于被管理的每种资源之中，相同供应商生产的中央服务器要连接每个代理，以管理资源。尽管大多数管理代理遵循简单网络管理协议(SNMP)标准，但大部分管理系统鼓励采用同构的管理基础设施，以实现最大的收益。这种方式导致采用广泛部署的大型系统管理框架，从而把客户的整体管理束缚于一家供应商。

当前系统管理正转向基于工业标准的Web服务，从而允许采用新一代模块化、互操作系统管理解决方案。在这种新型模式中，专用代理被使用XML和HTTP等开放标准而设计的代理取而代之，中央服务器被使用这些标准与代理执行交互的Web服务取而代之。

在此种模式中，系统与网络设备将为他们的设备提供管理代理，许多公司将竞相提供以特定管理领域为目标的模块化Web服务。因此，客户再也不会束缚于一家供应商，他们将能够为每个管理层面寻求最佳的供应商。

此种模式的标准被称为公用信息模式(CIM:Common Information Model)，是由坚持以分布式管理为重点的标准组织，即分布式管理特别工作组(DMTF)定义的。CIM为系统管理发挥的作用，跟ebXML为B2B交易发挥的作用相同。CIM旨在为设计Web服务，定义相关模式和协议标准。这些Web服务能够独立工作，跟它们交互的其他服务和代理的基本设计无关。例如，CIM为通过网络访问服务器基本信息，定义标准的XML对象，其中的许多对象已经在大量服务器操作平台予以实现，例如Solaris操作环境和Windows 2000。

此种模式在开放式智能Web服务领域产生了重要影响，表现在下列两个方面。

(1) Web服务将由此种模式执行管理。开发人员可以使用Java管理扩展(JMX) API改编Web服务，为管理Web服务的部署、健康和性能构建标准接口。CIM标准允许开发人员执行这项工作，他们是实现这些特性的最佳人选。当部署Web服务时，基于Web的管理服务将能够使用相同的标准查找这些服务，并通过其代理管理它们。

(2) 作为定义明确的服务，公司将能够把系统和网络管理外包给第三方管理服务提供商(MSP)，他们可通过网络执行这些任务。鉴于这些服务是模块化服务，客户可以逐项确定哪些管理任务需留给内部执行，哪些需实行外包。

8.4.3.5 智能过程

智能过程工具允许实现与用户信息相关服务的动作设计。基于用户信息的智能化处理技术，根据请求的用户信息，譬如所在地理位置、管辖范围或商务关系成熟与否等，以动态方式改变微观Web服务调用的设计顺序，进而改变宏观Web服务的结果。

在ebXML环境中，B2B商务过程表示为商务服务交换的设计顺序。每个基本服务交换都被称为商务交易。一般而言，商务服务是高级服务，例如订购服务或账单支付服务。但在幕后，一种商务服务通过循环方式，由许多低级服务组成，例如产品查找服务、报价服务和货币兑换服务等。

交易中心标记语言(XAML)为设计商务服务提供了一种可选的方法。XAML 计划是 Bowstreet、惠普、IBM、Oracle 和 Sun 联合发起的,宗旨是定义一系列 XML 信息格式与交互模式,以便协调和处理因特网上涉及多方的商务级交易。

8.4.3.6 智能化方针

Web 服务可以使用方针引擎,根据考虑用户身份、授权等级和其他背景信息的规则,以动态方式适应处理和/或结果。面向用户信息的标准仍有待定义,但许多与安全相关的标准和协议已经存在。用户和方针信息维护在一个采用 LDAP 协议可以访问的开放的目录中。Kerberos 与公钥基础构架(PKI)技术委员会的工作目标,就是为交换认证和授权信息定义一个 XML 框架。这样,就可以使用开放技术,以更加安全可靠的方式执行跨企业交易。该框架将为访问认证和授权服务提供请求/应答协议,用 XML 形式编写请求和确认身份与权利的代码,为各种传送与通信协议提供安全信息联编。OASIS 委员会把建议的两种安全技术规范,即安全服务标记语言(S2ML: Security Services Markup Language)与 AuthXML 合二为一。其中,S2ML 技术规范由 Bowstreet、Commerce One,Jamcracker、Netegrity、Sun、Verisign 和 Web Methots 等公司编写而成,AuthXML 技术规范由以 Outlook 科技公司为首的一个业界联盟编写。

8.4.4 智能化 Web 服务处理模式

智能化 Web 服务处理模式可以大致划分为 4 个步骤(图 8.4),它们分别是:服务准备、处理服务请求、服务工作流和交付服务应答。

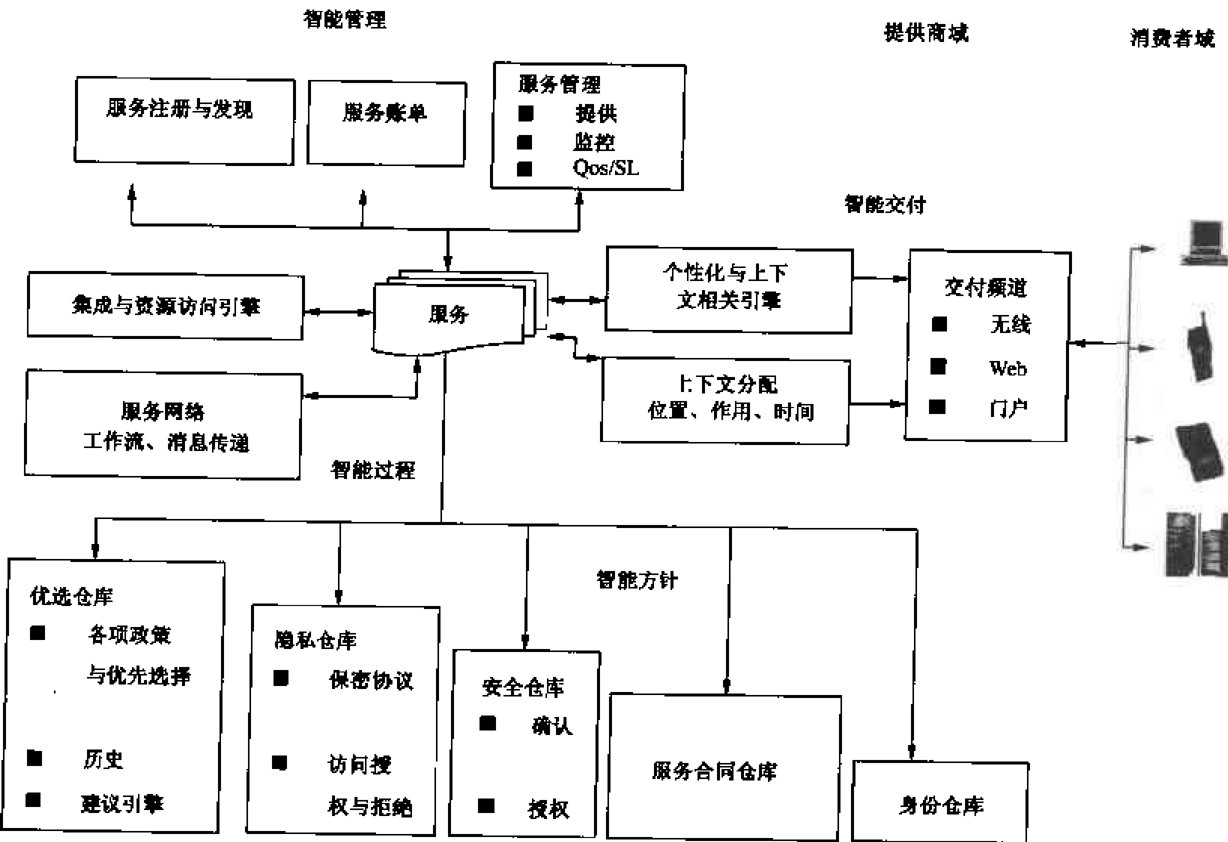


图 8.4 智能化 Web 处理

8.4.4.1 服务准备

智能管理工具旨在管理、监控和维护 Web 服务，其功能如下：

(1) 保证通过一个或多个服务注册器，正确地注册和定位服务。

(2) 保证签署和正确地执行相关使用收费或订购协议。

(3) 协调服务提供，并保证按服务级别协议规定的最低服务质量或其他标准执行服务。

(4) 从智能化方针工具确定管理和运行时方针。

智能化方针工具旨在管理和连接各种目录和仓库，其中包含的方针和商务规则将会影响服务的处理过程。

8.4.4.2 处理服务请求

服务请求可以通过任何类型的交付渠道提交。智能化交付工具根据用户地理位置、职务和要求服务的时间等信息，确定用户信息，并将用户请求传递给服务去执行。

8.4.4.3 服务工作流

智能化过程工具旨在管理宏观服务的动作设计。它根据用户信息和为服务定义的方针与商务规则，保证按正确的顺序调用相关微观服务或外部服务。服务使用集成和资源访问引擎调用其他服务，并访问数据库、文件、传统的应用和其他资源。

8.4.4.4 交付服务应答

处理完毕后，智能化交付工具使用个性化与用户敏感性引擎，为消费者量体定制应答。最终的应答通过相关交付渠道，交付给用户。

8.5 Web 服务开发模式

尽管可以使用任何编程语言开发 Web 服务，但是从现有的技术角度讲，主要是采用基于 J2EE(Java 2 Platform Enterprise Edition)标准或基于微软的.NET 产品系列。

8.5.1 基于 Java 的 Web 服务开发模式

J2EE 是由 SUN 引导，各厂商共同发起并得到广泛认可的工业标准。业内“企业计算”领域的大企业如：IBM，BEA Systems，Oracle 等都进行了参与。Sun ONE 软件构架可以看做是 J2EE 标准的一个典型应用，是基于 Java 平台的典型开发模式。Java 平台包括 XML 本机支持，Java API for XML Processing (JAXP)为连接 DOM、SAX 和 XSLT，提供一个 Java 接口。面向 XML 的其他 Java API，通过 Java Community Process Program，正处于不同的发展阶段。Java API for XML Data Binding (JAXB)把 XML 数据和 Java 代码相连接。开发人员使用 JAXB 把 XML 模式信息编译成 Java 对象。在运行时，JAXB 自动把 XML 文档数据变换成 Java 对象，反之亦然。Java API for XML Messaging (JAXM)为连接 ebXML MS、W3C XP 和 SOAP 等 XML 信息传递系统提供一个本机 Java 接口。Java API for XML Registries (JAXP)为连接 XML 注册器和仓库提供一个接口，例如 ebXML 注册器/仓库与 UDDI 商务注册器。一种新的 Java 技术规范请求已经提交 JCP，以便定义 Java API for XML RPC (TAX/RPC)，后者将为适用于 XML 信息传递系统的 RPC 编程常规提供

直接支持，例如 SOAP 和即将出台的 W3C XP。

Web 服务构成

图 8.5 从开发人员的角度显示微观 Web 服务。Web 服务由 Web 服务接口和一至两个服务部件组成。其中，Web 服务接口负责管理和操作 XML 信息。服务部件包括实现服务的商务逻辑，并经常通过一系列集成服务，跟外部资源与服务执行交互。

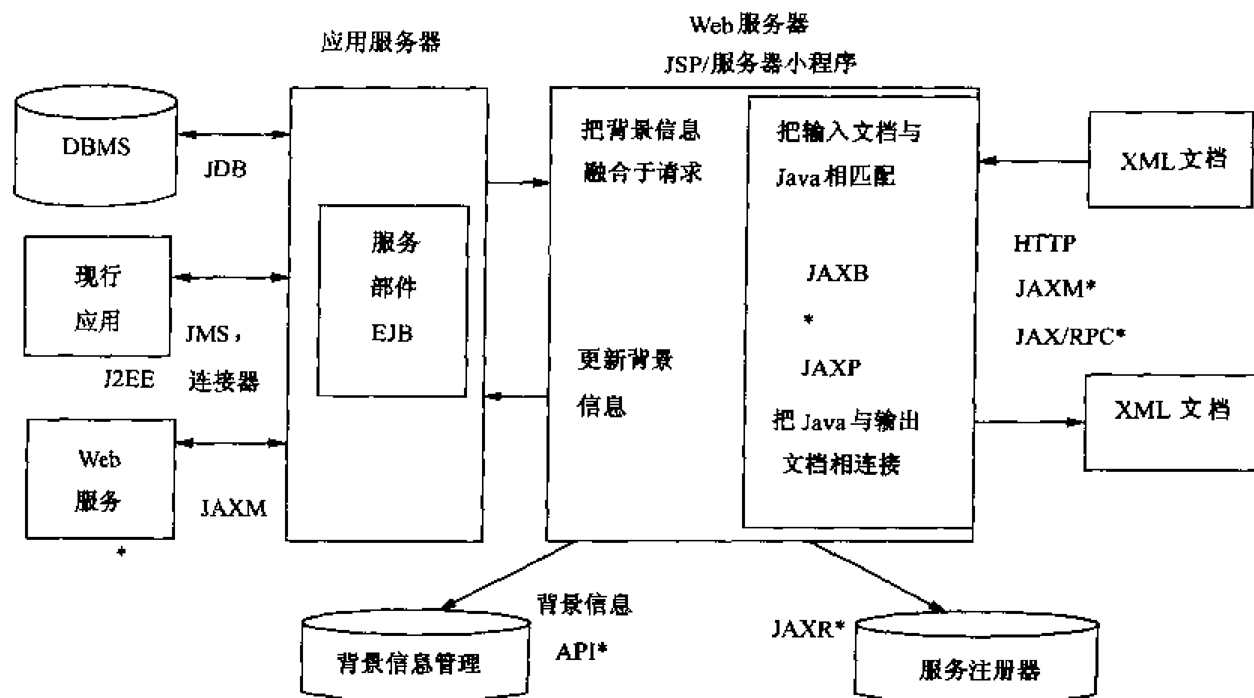


图 8.5　现在与未来的 Java

Web 服务接口

Web 服务通过标准 Web 协议，例如 HTTP，传送 XML 文档，由此实现通信。Web 服务接口实现生成并消费 XML 信息的代码。Sun ONE 软件体系结构建议采用 3 类 XML 信息传递系统：即 SOAP、ebXML MS 和将来的 W3C XP。

SOAP 是一种轻型可扩展 XML 信息传递协议。尽管 SOAP 是通用 XML 信息传递服务，但特别适合于 RPC 类服务调用。SOAP 信息是一种 XML 文档，由标题与正文组成。信息数据用 XML 详细说明，并封装于 SOAP 正文中。SOAP 支持 RPC 编程常规，后者将自动把输入、输出参数和 SOAP 信息正文里的 XML 元素相连接。由于 XML 文档不能包含另一个 XML 文档，故 SOAP 信息不能包含一个完整的 XML 文档，例如采购订单。SOAP 协议还不支持多媒体文件的附件或其他非 XML 数据。SOAP 不提供服务质量(QoS)框架。可以通过 Apache 软件基金会，获取实现 SOAP 接口的 Java 工具。基础 SOAP 环境可以扩展，以支持 QoS 框架，但现在还不能实现。

ebXML MS 是一种 XML 信息传递服务，是为了支持 B2B 商务需求而设计的。ebXML 信息是一种多方/MIME 封装信息，可以传输任何数量的 XML 文档和非 XML 附件。cbXML 不支持 RPC 编程常规，但支持 QoS 框架以保障可靠地交付信息。QoS 框架可以扩展，以支持安全与交易语义。ebXML 技术规范于 2000 年 5 月出台。

W3C XP 技术规范处于开发之中。关于 W3C 的工作进展细节未向外界透露。Web 服务接口是作为运行 Web 服务器内的 JSP 页面或服务器小程序予以实现的。JSP 页面/服务器小程序接收 XML 信息,并析取 XML 文档。当前,该过程以人工方式执行。将来开发人员可以使用 JAX 或 JAX/RPC,以更有效的方式处理 XML 信息,JSP 页面/服务器小程序接收 XML 文档,把文档数据变换成 Java 对象数据。今天,开发人员可以使用 JAXP 处理 XML 文档,将来可使用 JAXB 自动把文档和 Java 对象相连接。

捕捉背景信息

Web 服务接口还尽可能地捕捉许多背景信息。通过检索客户机系统的 Cookie 目录,就可以捕捉用户身份,或者要求用户提供用户 ID 与口令。一旦服务接口确定了身份,服务就可以查询专用用户历史文件,以检索附加背景信息。当前,Sun 与伙伴携手推进一个标准项目,为处理身份和用户信息定义标准框架。用户背景信息框架的细节尚未定义,但预计将有标准的 API 推出。届时 JSP 页面/服务器小程序可以使用这些 API,获取用户的背景信息。

商务逻辑

一旦 JSP 页面/服务器小程序把 XML 数据变换成 Java 数据,并捕捉了用户背景信息,便可以执行通常的会话管理服务,并调用商务逻辑,由商务逻辑执行实际服务。商务逻辑是作为 EJB 部件或适用于轻型应用的服务器小程序予以实现。商务逻辑部件将与管理整合商务交易的工作流引擎相连接,并与根据用户身份和身份背景,实施相关商务规则的方针引擎相连接。目前,该过程的标准尚未定义。但 ebXML、XAML 和 OASIS 安全服务将在该过程中发挥作用。

集成服务

商务逻辑部件将需要连接各种资源。JDBC API 提供数据库访问,J2EE 连接器与 Java 信息服务(JMS)提供其他应用系统访问,XML 信息用于连接其他 Web 服务。当前,开发人员需要使用 JAXP 以人工方式构建 XML 信息。将来,开发人员可以使用 JAXM 或 JAX/RPC,以更便捷的方式操作 XML 信息。

回送结果

当商务逻辑部件完成工作后,将向 JSP 页面/服务器小程序接口部件回送 Java 结果对象。接口使用 JAXP,把结果从 Java 对象变换成输出 XML 文档,将来可以使用 JAXB。然后,JSP 服务器小程序可以使用背景信息,进一步应答文档。定制完成后,应答将回送给请求者。

8.5.2 微软.NET 开发模式

微软的.NET 是一个产品的序列,大部分是对微软早期的企业开发平台 Windows DNA (Distributed internet Applications Architecture)的重写。Windows DNA 包括 Microsoft Transaction Server(MTS),COM+,Microsoft Message Queue(MSMQ)以及 Microsoft SQL Server 数据库。新的.NET 框架对这些技术进行了改进,这包括增加了一个 Web 服务层以及增强的语言支持。微软.NET 用于构造 Web 服务的开发者模型如图 8.6 所示。

.NET 应用嵌入在一个容器中,该容器提供了企业应用必须的服务质量,这些包括事务、安全和消息服务。

商务层执行商务处理和数据逻辑,使用.NET 管理的部件进行构造。该层使用 Active Data Objects(ADO.NET)连接数据库,使用 Microsoft Host Integration Server 2000 提供的服务与现有的系统进行连接,这些服务包括 COM Transaction Integrator(COM TI)。该层使用 Web 服务的技术,如 SOAP,UDDI 和 WSDL 与其他的商业伙伴进行连接。

商务伙伴使用 Web 服务的技术,如 SOAP,UDDI,WSDL 和 BizTalk 与.NET 进行连接。

传统的用户、Web 浏览器和无线服务以 HTML,XHTML 或 WML 等标准嵌入在 Active Server Pages(ASP)与.NET 进行连接。需要额外进行处理的用户接口使用 Windows Forms 构造。

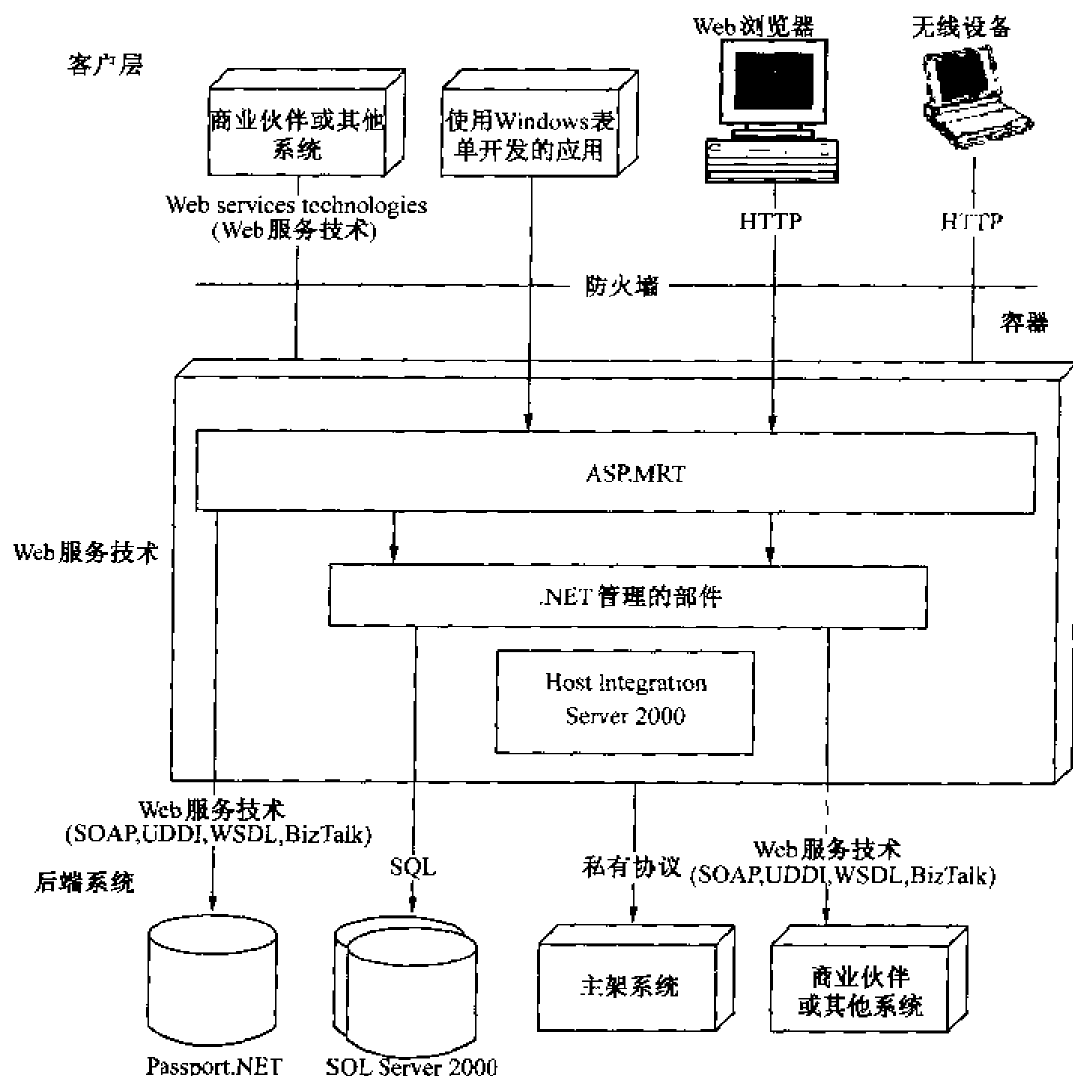


图 8.6 使用微软.NET 开发 Web 服务

参 考 文 献

1 (美)Didier Martin 等著. XML 高级编程. 严春莹等译.北京:机械工业出版社,2001

2 孙一中等编. XML 理论和应用基础. 北京:北京邮电大学出版社, 2000

3 Resource Description Framework (RDF) Model and Syntax. W3C Recommendation, 22 February 1999. http://www.w3.org/TR

4 Resource Description Framework(RDF) Schema Specification 1.0, W3C Candidate Recommendation 27 March 2000, http://www.w3.org/TR

5 罗威. RDF——Web数据集成的元数据解决方案. 计算机世界,2001.6.8

6 Guarino. N, Giaretta P. Ontology an Knowledge Bases towards a terminological clarification. Towards very Large Knowledge Bases —Knowledge Building and Knowledge Sharing, ISO Press,1995. 25~32

7 Genesereth M R, Nilsson N J. Logical Foundation of Artificial Intelligence. San Mateo, CA: Morgan Kaufmann Publishers, 1987

8 Fernández M, Gómez Pérez A, Juristo N. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. Spring Symposium Series, Stanford, 1997

9 Gmez-Prez A, Fernndez M, De Vicente A. Towards a Method to Conceptualize Domain Ontologies. Working notes of the workshop Ontological Engineering. ECAI'96: 41~52

10 Thomas R Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University

11 MIKE USCHOLD, MARTIN KING, STUART MORALEE, YANNIS ZORGIOS. The Enterprise Ontology. The Knowledge Engineering Review, 1998.13(1):31-89

12 Guarino N, Welty C.(2000c). Towards a methodology for ontology based model engineering. In Proceedings of ECOOP-2000 Workshop on Model Engineering. Cannes, 2000

13 Uschold, M, Gruninger, M. ONTOLOGIES: Principles, Methods and Applications. Knowledge Fngineering Review. 1996. 6,11(2)

14 Speel H, Raalte F. Scalability of the Performance of Knowledge Representation Systems. Towards Very Large Knowledge Bases. IOS Press. Amsterdam. 1995.173~184

15 Strawson P F. An Essay in Descriptive Metaphysics. Routledge, London and New York. 1959

16 Lowe E J. Kinds of Being, A Study of Individuation, Identity and the Logic of Sortal Terms. Basil Blackwell, Oxford. 1989

17 Harold B, Stefan D, Michael S. Tutorial n Knowledge Markup Techniques, ECAI

2000 Berlin 2000.8.22
18 Harold B, Stefan D, Michael S. Tutorial n Knowledge Markup and Resource Semantics, IJCAI-01, Seattle, 2001.8.6
19 规则标记语言 RuleML. http://www.dfki.uni-kl.de/ruleml
20 知识标记技术. http://www.dfki.uni-kl.de/km/knowmark
21 Simple HTML Ontology Extensions(SHOE). http://www.cs.umd.edu/projects/plus/SHOE
22 Relational-Functional Markup Language. http://www.relfun.org/rfml.
23 Boris Lublinsky. XML and Distributed Computing. XML journal, 2(4),8～17
24 Nick Simha, Dermot Russel. Building Distributed Applications with CORBA and XML. XML journal. 1(1),42～57
25 (美)Otte R,Partrick P, Roy M. CORBA教程——公共对象请求代理体系结构.李师贤等译校.北京:清华大学出版社,1999
26 Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000. http://www.w3.org/TR
27 Ambler S W, 1995b. Mapping Objects to Relational Databases. Software Development, 1995.10.63～72
28 Ambler SW. Object-Relational Mapping. Software Development. 1996. 10.47～50
29 J2EE vs. Microsoft.NET — A Comparison of building XML based web services. http://www.middleware-company.com/
30 Microsoft .NET White Paper. http://www.microsoft.com/net/
31 SunTM Open Net Environment(ONE) Software Architecture. http://www.sun.ca/software/sunone/wp-arch/wp-arch.pdf
32 孟小峰. Web数据管理研究综述.计算机研究与发展. 2001第1期
33 徐振航,刘莉芹.XML与面向Web的数据挖掘技术.中国计算机报.2000.10(1)
34 丁茂良. 信息集成浅论. 网络世界.2000.4(13)
35 黄宏斌等. 基于关系数据库的数据仓库框架设计.全国计算机新科技与计算机继续教育论文集.1999
36 徐振宁,张维明,陈文伟. 基于Ontology的智能信息检索.计算机科学.2001(28):6.
37 Grady Booch, Ivar Jacobson, James Rumbaugh. Unified Modeling Language. RationalSoftware Corporation. Version 1.0. 1997.1
38 Matthew Fuchs. Schema for Object-oriented XML. W3C, 1998. Seehttp://www.w3.org/Sumission/1998/15
39 Charles Frankston, Henry S Thompson. XML Data Reduced. http://www.ltg.ed.ac.uk/～ht/XMLData-Reduced.htm
40 Document Object Model. http://www.w3.org/
41 Simple API for XML. http://www.megginson.com/SAX, http://www.megginson.com/SAX/SAX2.
42 Rational, UML for XML Schema Mapping Specification. http://www.rational.com/

media/uml/resources/media/uml _ xmlschema33. pdf.
43 OMG. OMG UML 1. 3 UML Semantics. http://www. omg.org
44 OMG. OMG UML 1. 3 UML Summary. http://www. omg. org
45 OMG. OMG UML 1. 3 UML GLOSSARY. http://www. omg. org
46 OMG. OMG UML 1. 3 UML Notation. http://www. omg. org
47 Rational Software. UML Extension for Objectory Process for Software Engineering. http://www. rational. com/uml